ZORAN

Vector Signal Processors

2.35

ZORAN

Vector Signal Processors

ZR34161

VECTOR SIGNAL PROCESSOR SIMULATOR

USER'S MANUAL

July 1986

For Beta-site Simulator Release 2.35

Zoran Corporation

3450 Central Expressway

Santa Clara, California 95051

The work on which this manual is based was supported in part by a grant from the Israel-United States Binational Industrial Research and Development Foundation. The views and information contained herein are those of the authors and not necessarily those of the Foundation. The Foundation assumes no liability for the contents of this document by virtue of the support given.

"Man is a tool-using animal...
Without tools he is nothing, with tools he is all."

Thomas Carlyle

CONVENTIONS

For ease of reading, the Zoran Vector Signal Processor will be referred to in this manual as the *VSP*. The Vector Signal Processor Simulator will be referred to as the *VSPS*. Fast Fourier Transform will be abbreviated *FFT*.

Please keep in mind:

As with any large software project, continuous updates and enhancements are being made. Please return the enclosed card to ensure receiving all updates and new releases of the software manual and simulator. In addition, Zoran welcomes (and encourages) notification of any errors, problems or omissions detected in using this manual or the software simulator. Any additional suggestions for improvement are always welcome.

This latest update reflects modifications made to several of the menus within the VSP Simulator. The basic functions of the VSPS have not changed, although a few new options have been added. Please refer to Chapter IV for these menu modifications.

Thank you for your interest and support of Zoran, and good luck with your designs.

ERRATA SHEET FOR VSPS-161 SIMULATOR MANUAL BETA-SITE VERSION 2.35

4 August 1986

There is currently an unfortunate, yet unavoidable confusion existing in the simulator and its manual regarding the use of the FSIZ parameter in the DEMO and MODLT instructions. The table below highlights this ambiguity. When programming logical FSIZ values with these instructions, the *current logical value* must be specified. The value used by the simulator is the *actual logical value*. The simulator internally carries out this translation. If values are specified as literals, the simulator will translate the literal value to the *actual logical value*. For instance, selecting a *logical* FSIZ parameter of 16 in the simulator will generate an *actual logical* FSIZ *value* of 8, with the corresponding literal value of 001. It is not possible to select an actual logical value of 64.

FSIZ - Specifies the number of samples beginning with RBA to be addressed from the internal sine/cosine LUT, after which the LUT address rolls back to the RBA value.

Literal Value		**Current Logical Value**		**Actual Logical Value**
000	->	8	->	4
001	->	16	->	8
010	->	32	->	16
011	->	64	->	32
100	->	128	->	128

The above table should be used in place of the tables on pages 6-38 and 6-41 in the simulator manual for beta-site release version 2.35.

TABLE OF CONTENTS

LIST OF FIGURES

APPENDICES

CHAPTER I

INTRODUCTION

1.1 Overview

The ZR34161 Vector Signal Processor is a very powerful single-device processor designed for solving computation-intensive digital signal processing applications. Because it is a general-purpose programmable peripheral, it requires a control program which provides instructions for execution, and because it is primarily a high-performance digital signal processor, it is necessary to provide an environment which allows for algorithm development and simulation prior to final implementation in hardware. In order to speed system and algorithm development time, the *Vector Signal Processor Simulator (VSPS)* was created.

The VSPS is a software tool which provides arithmetic, I/O and timing modeling of the VSP device. However, the VSPS is not simply limited to modeling the VSP hardware functions. Also included is the ability to model *the world external to the VSP as well as the VSP itself.* This means that the VSPS provides the ability to simulate the signal processing algorithms as they will be executed within the VSP, as well as modeling instruction fetch, bus access and bandwidth utilization, host control operations, and ultimate system timing. Multiple VSPs in a single application can also be simulated using the VSP simulator. The goal of the simulator is to allow the majority of system development to be performed in software instead of in hardware.

A number of development tools are provided within the simulator environment, including:

- an instruction tutorial including single-step instruction execution,
- a signal generator,
- terminal plotting capabilities,
- a macro command-language,
- IEEE signal processing options.

1.2 Purpose

The purpose of this manual is:

- to familiarize users with the Zoran Vector Signal Processor (VSP), a monolithic array processor;
- to teach users how to use the Zoran Vector Signal Processor Simulator (VSPS), a software program simulating the functions of the hardware;
- to teach users how to program the Zoran Vector Signal Processor and its host using the VSP language;
- to serve as a reference guide after learning to use the VSPS.

1.3 Scope

The scope of this manual is to provide a complete description of the VSPS and its usage. This includes the on-screen tutorials, interactive use of the VSPS utilizing both menus and the macro command-language, all of the facilities and tools provided within the simulator environment, programming the VSPS using the VSP language, and use of the VSPS compiler.

1.4 The User

This manual is written for the user with some technical background. General knowledge and familiarity is assumed in the following areas:

- computer basics and terminology;
- fundamentals of computer programming;
- at least one field of application in digital signal processing;
- the general use of the Fast Fourier Transform to convert digitized data into different domains;
- at least one advanced programming language, such as C, Pascal or Fortran.

1.5 Manual Overview

Chapter II, *The Zoran Vector Signal Processor*, briefly covers the VSP with enough information about the device to provide the background needed for understanding and using the simulator. This coverage does not require an engineering background. More thorough and detailed technical information about the hardware exists in the VSP engineering data sheet.

Chapter III, *The Vector Signal Processor Simulator*, presents an overall look at the VSP simulator so that maximum benefit can be gained from the "how to" chapters that follow.

Chapter IV, *VSPS Interactive Usage*, covers the aids to learning, modes of operation, and the menus as a way to use the simulator.

Chapter V, *VSP Registers*, explains the operation and use of the hardware registers in performing computations with the VSP.

Chapter VI, *VSP Instruction Set*, discusses the VSP instruction set and associated parameters in detail. Examples of all instructions are provided in this chapter.

Chapter VII, *Macro Command-Language*, is a description of the simulator execution environment which allows users to bypass the menu structure of the VSPS and enter commands directly.

Chapter VIII, *VSP Applications*, describes some of the applications in which the VSP excels, and explains the way in which the VSP may be programmed for certain applications.

Chapter IX, *Programming the VSPS*, explains in detail how to write, parse, compile, link and execute VSP programs using the simulator.

Chapter X, *Simulating the Entire VSP/Host System*, describes the hardware configuration of the VSP and host as modeled by the VSPS, as well as the software operating system the VSPS assumes in the host.

The appendices contain supplementary technical information, simulator installation procedures and reference material for use after familiarity with the VSPS has been gained.

1.6 How to Use this Manual

To understand the material in this manual, it is suggested that the chapters be scanned in the order presented. Then, depending on the user's skill level or particular areas of interest, specific chapters can be concentrated on.

After familiarity has been gained with the VSPS and the contents of this manual, the Table of Contents and appendices will be found to be quite useful for quick reference.

CHAPTER II

THE ZORAN VECTOR SIGNAL PROCESSOR

2.1 Overview

This section is intended as an introduction to the ZR34161 Vector Signal Processor. Included are general discussions of the features, hardware and software interfacing and certain performance benchmarks. It is not intended as a comprehensive description of the device, its pinout or architecture. For detailed information in these areas, please refer to the VSP engineering data sheet.

2.1.1 What Is the Vector Signal Processor ?

The Zoran Vector Signal Processor (VSP) is a special-purpose programmable digital signal processor optimized for performing vector types of signal processing operations. A unique proprietary architecture allows the VSP to achieve extremely high data throughput rates previously achieved only by board-level designs. The VSP contains a powerful instruction set, simplifying the programming effort required to implement the user's target application. The result is a system that is extremely powerful for implementing high-performance digital signal processing applications while maintaining simplicity in hardware interfacing and software development. The VSP is built in CMOS technology and is housed in a 48-pin DIP package.

2.1.2 What Are Its Functions ?

The Zoran Vector Signal Processor:

- performs Fast Fourier Transforms and other vector operations specifically tailored for signal and image processing.
- executes a high-functionality set of 23 instructions.
- can perform FFTs of size up to 128 complex points in one instruction without external RAM; of size up to 1024 complex points without external sine/cosine tables; and of size up to 64K complex points maximum size.
- moves blocks of data to and from external memory in a single instruction, using a 16-bit address bus and a 16-bit data bus.
- can be used in parallel with multiple VSPs in single applications for even higher throughput.

2.2 Description

2.2.1 General

Figure 2-1 shows the pin assignment diagram and pin names for the ZR34161 VSP. It is housed in a 48-pin ceramic dual in-line package (DIP). The device has a 16-bit address and 16-bit data bus. The bus interface, including control signals, is designed to interface easily with any host microprocessor or system controller. Detailed descriptions of the pins are provided in the VSP engineering data sheet.

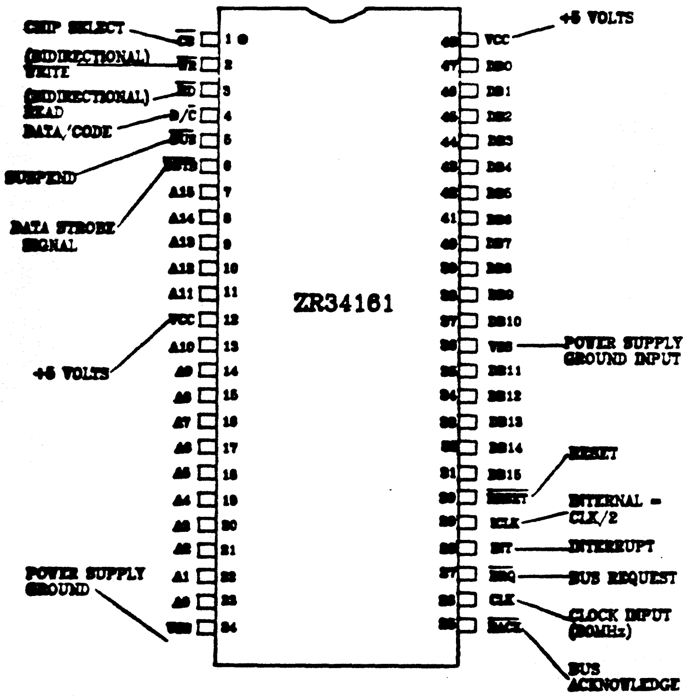


Figure 2-1. Pinout of the ZR34161 Vector Signal Processor.

2.2.2 Architecture

Figure 2-2 shows a block diagram of the architecture of the VSP. It is comprised globally of three main blocks: the bus interface unit (BIU), the execution unit (EU), and the memory and registers. The bus interface unit is effectively everything on the left hand side of the figure. Included in the BIU are:

- data bus buffers,
- an address generator,
- an instruction fetch unit,
- a bus control interface.

The BIU is responsible for executing all bus-related operations including data and instruction I/O, bus timing, instruction fetch and decoding, and communication among the internal and external memory devices. DMA activities involving data and instruction fetch are also controlled within the BIU. An instruction FIFO (first-in, first-out buffer) is present in the instruction fetch block which is able to store up to four VSP instructions.

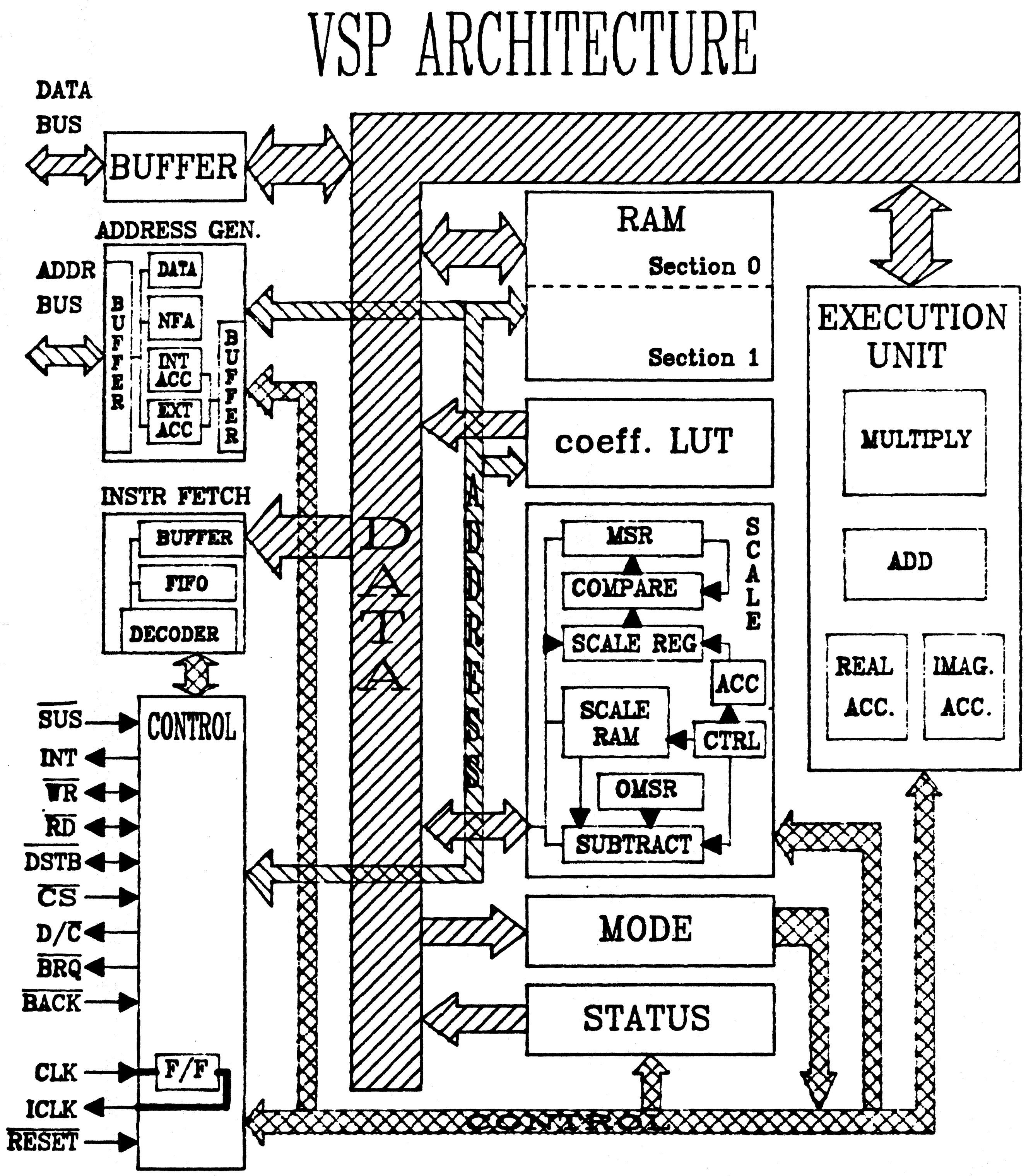


Figure 2-2. Internal VSP Architecture.

The execution unit is responsible for all ALU-intensive operations. It is made up of:

- a 17 x 17-bit multiplier,
- an adder,
- two 24-bit accumulators.

Coupled very tightly to the execution unit and the bus interface unit are memory and registers, consisting of:

- a 128 x 38-bit data RAM,
- a 64 x 4-bit scale RAM for implementing block floating-point operations,
- a 256 word by 17-bit sine/cosine look-up table,
- operating mode and status registers.

The RAM contains 256 19-bit words configured as 128 complex words. All arithmetic operations performed by the execution unit on external data first pass through the bus interface unit. The FIFO, scale RAM, registers and look-up tables, as well as the control unit, are all shared by both the Bus Interface Unit and the Execution Unit.

The RAM can be configured into two independent sections, each consisting of 64 complex words or less. Each complex word is made up of one 19-bit real part and one 19-bit imaginary part. One of the independent RAM sections can be accessed by the BIU and the second section by the EU. This powerful feature allows I/O to be performed *nearly concurrently* with ALU operations. For example, assume that an application requires continuous real-time FFT calculations. While the butterfly calculations of one FFT are in progress in the execution unit, the BIU may be storing the previous FFT calculation to external memory and reading in the data to be transformed next.

NOTE: ***Technical details of concurrent I/O with ALU execution are provided in the VSP engineering data sheet.***

The VSP also contains a number of registers, shown in the logical architecture diagram in Figure 2-2(b), with names and bit lengths as follows:

Register Name	# Bits
Mode	16
Status	16
Instruction FIFO	192(4 three-word instructions)
Instruction Base/Start	16
Next Fetch Address	16
Scale	16
Maximum Scale	4
Old Maximum Scale	4
Real Accumulator	24
Imaginary Accumulator	24

2.3 Instruction Set

The instruction set of the VSP is designed to be highly functional; each instruction can be thought of as analagous to a subroutine kernel in a signal processing library. The VSP is programmed at the *functional* level, not the traditional assembly level required of other signal processing components. Because of the high level of functionality provided by each instruction, the VSP uses a low percentage of the bus bandwidth for instruction fetch. This also simplifies the programming and debugging time required to implement signal processing algorithms. For instance, the command to perform a Fast Fourier Transform, "*FFT*", is a single instruction within the VSP.

The 23 instructions within the VSP are categorized into four functional types as follows:

1) move blocks of data in either direction between the VSP internal memory or registers and external memory:

   LD (Load)
   ST (Store)
   LDSM (Load Scale/Mode Reg)
   STB (Store Backward)
   STI (Store Information Reg)

2) issue ALU/memory instructions with two vectors as operands, one residing in the internal VSP RAM and the other residing in external memory:

   ADDR (Vector Add Real)
   ADDC (Vector Add Complex)
   MLTR (Vector Multiply Real Accumulate)
   MLTC (Vector Multiply Complex Accumulate)

3) issue ALU instructions which operate on a single vector stored internally in the VSP RAM:

   ACCR (Accumulate Real)
   ACCI (Accumulate Imaginary)
   ABS (Absolute Value)
   CMCN (Complex Conjugate)
   DEMO (Demodulate)
   MODLT (Modulate)
   SCL (Scale)
   SCLT (Scale Literal)
   FFT (Fast Fourier Transform)
   CMLT (Cross Multiply Accumulate)
   MGSQ (Magnitude Square Accumulate)

4) control instruction fetch and execution of the VSP:

JMPI (Jump Indirect)
HLT (Halt)
NOP (No Operation)

Most of the instructions have several parameters, the values of which control the way the instruction is executed. Full treatment is given to the instruction set in Chapter VI.

All the instructions in the first two groups above consist of three 16-bit words. Those in the third group vary in length from one to three words. The No Operation (NOP) instruction is one word, and the Halt instruction is two words in length.

2.4 Block Floating-Point Operation

The ZR34161 Vector Signal Processor is a 16-bit integer machine. However, it also possesses the ability to perform *block floating-point* operations. Block floating-point capability allows the association of an exponent with a *block* of data. A block of data in this discussion is a real, imaginary or complex vector (array) of data stored inside the VSP. The entire block (or portions of the block) can then be scaled by this exponent. This differs from full floating-point precision in that full floating-point associates an exponent with *each* data value which is represented. The block floating-point capability is extremely powerful, especially when performing FFTs or related operations; it can increase the dynamic range of the FFT by up to 48 dB relative to a 16-bit integer FFT calculation.

2.5 VSP/Host Interface

The VSP communicates with a host computer or controller over a familiar and simple interface bus. An example of this interface is shown in block diagram form in Figure 2-3. The system bus contains separate address, data and control signals. This interface provides the necessary communication between the host and VSP for instructions, data, addressing and control signals. Oftentimes the VSP system controller will be a microprocessor, with the VSP operating as a slave peripheral.

Moreover, the VSP has the ability to operate with more independence than simply a slave peripheral. If the VSP is operated in the *master fetch mode*, it has the capability of fetching its own instructions and data after it has been given a starting address by either a controller or a host microprocessor. It is not required in this application that a sophisticated host microprocessor be present for system control. Any controller or state machine may be used to write a starting address to the VSP.

The system usually contains program memory which may be ROM, PROM, EPROM, etc.. This memory may exist on the main system bus within the system memory address space as shown in Figure 2-3, or it may be on a private bus belonging solely to the VSP. The VSP will maintain pointers to this memory for fetching instructions, as illustrated by the *Instruction Base Address* and *Next Fetch Address* pointers.

The same discussion is true for the VSP data RAM. It may exist within the address space of the host system as shown in Figure 2-3, or it may exist on its own private bus. The RAM can be partitioned into different memory block sizes as illustrated. The *Memory Base Address* is a pointer specified by VSP instructions which use external data memory.

A bus arbitration block normally exists in a single-bus system such as that shown in Figure 2-3 in order to resolve conflicts which may occur over bus access requests. The VSP has a simple two-pin interface to the bus arbiter for requesting and receiving control of the bus. A simple yet powerful DMA controller exists on the VSP for easy interfacing to external ROM and RAM once bus access is granted to the VSP by the host. Instructions, status and data are all passed over the 16-bit data bus using this DMA structure. Data and instruction addressing is generated on the 16-bit address bus.

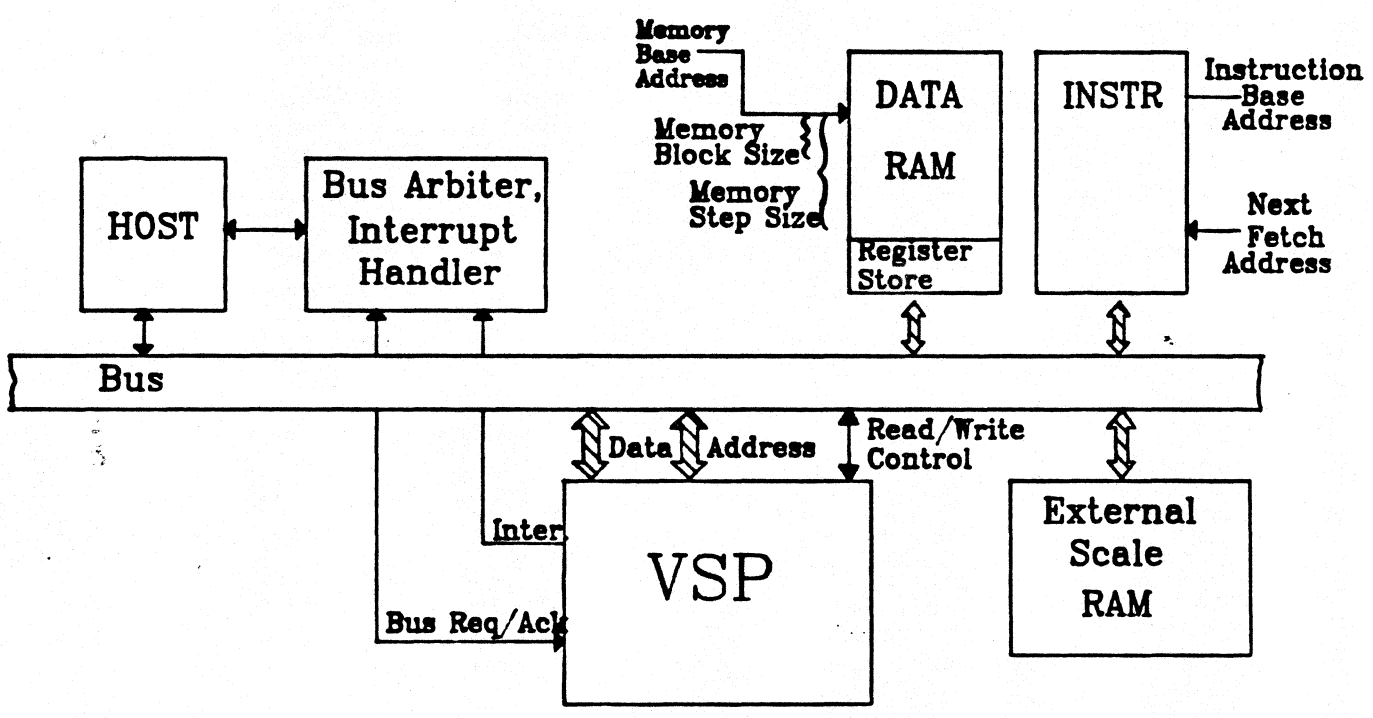


Figure 2-3. Block Diagram of the VSP/Host System.

2.6 Addressing Modes

The VSP receives instructions for execution in one of two ways:

Master Mode: The VSP fetches its own instructions. The host writes the starting address of the instructions to the VSP, which then fetches its own instructions, fills its instruction queue and begins execution. It will continue instruction fetch and execution until a HALT instruction is encountered.

Slave Mode: The host loads the VSP with instructions as execution is required. The host may load the VSP with instructions either without execution (which loads the on-board FIFO) or with a command to execute the instruction immediately.

When the VSP is fetching its own instructions and data, it requests and receives control of the bus, then generates memory addresses for reading and writing data to and from external memory.

The two instruction addressing modes allow a great deal of flexibility for different system configurations. In the master mode, the VSP simply needs to be told where to fetch the first instruction in memory. It will then fetch all of its own instructions and data beginning with that location. In this configuration, the VSP can be controlled by a device as simple as a single-chip microprocessor or state machine.

In a host microprocessor controlled environment where the VSP is operating in the "master" mode, the host could send starting addresses of "subroutines" to the VSP. The VSP would then begin reading and executing instructions beginning at the "subroutine" address and continuing until a "Halt" instruction is encountered. This application essentially uses the VSP as a loosely-coupled co-processor. The VSP would perform all of the signal processing required in the application under direct control of the host processor.

2.7 Example System Configuration

Figure 2-4 shows a potential example system incorporating the VSP. The system manager in this application is a simple state-machine controller, not a full microprocessor. The controller has the ability to interpret simple status appearing at its input pins, arbitrate access to the data bus, write the beginning program address to the VSP, and manage other peripherals (not shown in the figure) which may be connected to the bus. The non-volatile memory contains the executable VSP instructions. The RAM acts as data storage as well as additional scratch-pad memory for the VSP. It is assumed that there may be other peripherals connected to the data bus which provide data to the VSP for processing and also receive processed data from the VSP. The state-machine controller also manages these devices.

All data is exchanged between the components in the system using the 16-bit address and 16-bit data buses. Familiar control signals such as RD\, WR\, CS\, SUS\, and DSTB\ are present to coordinate this data flow. The VSP requests control of the data bus with the BRQ\ pin. Bus control is granted to the VSP on the BACK\ pin from the host controller. The D/C\ pin is provided to inform external memory of whether the data being fetched is operand data or instructions.

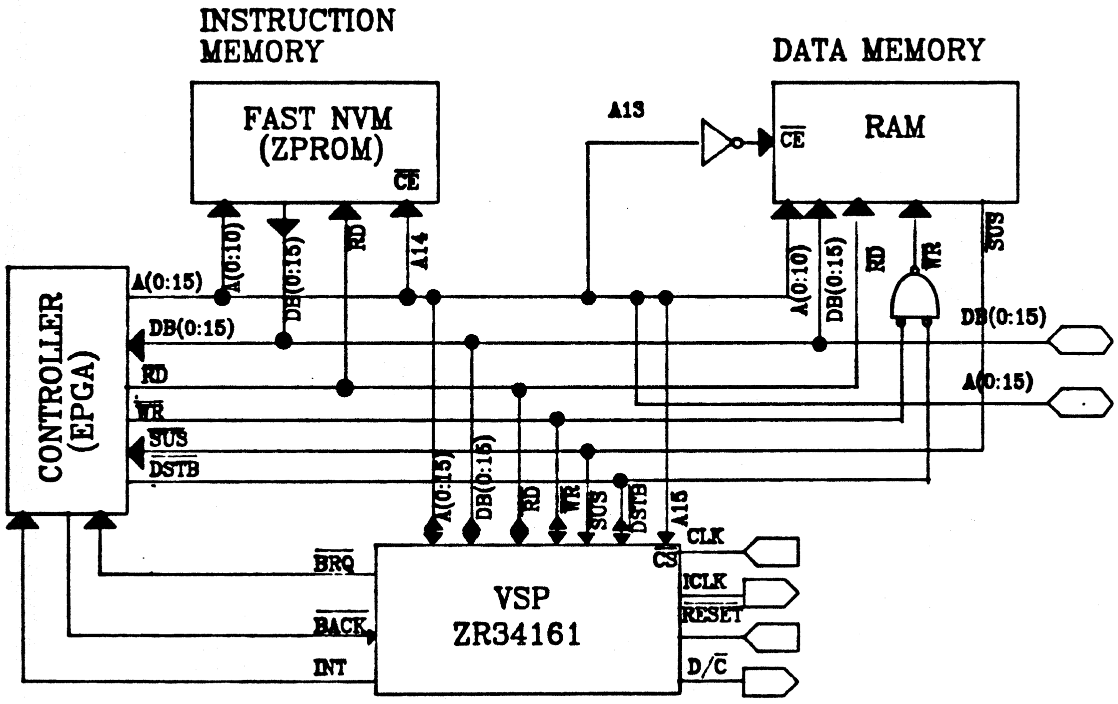


Figure 2-4. VSP System Under Simple State-Machine Control.

2.8 Performance Highlights

The VSP uses an external crystal or clock up to 20 MHz. This clock is divided by two internally to 10MHz, which represents the 100ns execution cycle of the VSP. The military version of the VSP operates with a 16MHz external clock which corresponds to a 125ns execution cycle. Table 2-1 shows some benchmark calculation times for certain signal processing operations within the VSP. These tabulated calculation times assume that the VSP is being driven with a 20MHz external clock.

"BF" in the FFT calculation refers to *block floating-point* calculations. Integer FFT calculations require even less time to complete. The 1024-point FFT requires reads and writes to external memory in order to accomplish the complete FFT because the VSP has an internal memory size of 128 complex words. *The 3300us includes all external reads and writes to memory*. It is assumed that the external memory has an access time of less than 100ns. For slower memory, with an access time between 100ns and 200ns, it will take the VSP 3700us to complete the 1024-point complex transform.

Table 2-1

Signal Processing Operation	**us**
1024-point complex BF FFT	3300
128-point complex BF FFT	237
8x8 2-D complex FFT	164
64x64 complex vector multiply	27
64-point complex demodulation	26
128-point magnitude square/accumulate	26
4x4 matrix multiplication	33

CHAPTER III

THE VECTOR SIGNAL PROCESSOR SIMULATOR

3.1 Overview

The *Vector Signal Processor Simulator (VSPS)* is a menu-driven software system designed to model both the hardware and software operation of the Zoran Vector Signal Processor. The purpose of the simulator is to allow full algorithm and target system simulation within the environment of the VSP simulator.

Instruction execution from within the simulator environment occurs in either of two ways. The first of these, *interactive execution*, provides on-screen menus to allow interactive execution of individual VSP instructions. The second, *programmed execution*, allows execution and testing of a complete program written with the VSP instruction set embedded within a high-level 'C' program.

The VSP simulator contains a powerful alternative to its menu-driven structure for more advanced simulator users, called the macro command language. The *macro command-language* is an environment where the menus are bypassed and the user can execute macro-level commands directly without traversing the menus. A library of macro commands is provided by the simulator; users are also encouraged to create their own macros and include them within this library. Full treatment of the macro command-language is given in Chapter VII.

The VSPS accurately simulates the execution of instructions by the VSP. Once an algorithm has been defined, it may be programmed into the simulator and its execution verified. Facilities are available within the VSPS environment for the purpose of simplifying algorithm debugging. For example, floating-point IEEE signal processing algorithms are provided to help compare signal processing results obtained with the VSP to floating-point results. Test and noise signals may be generated from within the VSPS for input to the simulation. The simulator also supports graphical output modes for display of waveforms at user-selected memory locations within the system. HELP utilities are provided containing descriptive material and explanations of menu options, instructions and processes.

3.2 Purpose And Functions

The VSPS has three major purposes:

- to provide a detailed interactive demonstration tutorial for each VSP instruction.
- to accurately simulate the hardware performance of the VSP for algorithm verification, speed and arithmetic accuracy.
- to simplify the task of developing VSP programs.

The functions of the VSPS are as follows:

- generation of sophisticated test waveforms;
- simulation of VSP programs;
- simulation of host programs running with VSP programs;
- simulation of single or multiple VSPs on a single bus;
- plotting of waveforms;
- interactive debugging;
- timing of VSP programs and measurement of effects of bus usage;
- modeling VSP arithmetic;
- provision of a macro command-language interface;
- provision of user-friendly, interactive interface.

3.3 Host Computer Requirements

Version 2.3-5 of the VSP simulator is written to operate under either the VMS or ULTRIX operating systems on the Digital Equipment Corporation VAX family of computers, or on an IBM PC/XT or PC/AT (or compatible) under DOS version 2.1 or higher. Both VMS and ULTRIX operating systems are supplied and supported by DEC. ULTRIX is the DEC version of UNIX 4.2bsd. A 'C' compiler and linker are required under either operating system. DOS is the operating system developed by Microsoft Corporation for the IBM PC family of computers.

3.3.1 VAX Computers

VAX computer requirements include about one megabyte of user memory in the host computer as well as a disk file system with at least one megabyte of user disk space. The terminal should be a DEC VT240 or VT100 (or compatible). The simulator can make use of the expanded capabilities of the VT240 terminal. Use of this or another compatible terminal allows high-resolution plots of waveforms within the simulator environment, as well as "paged" menu operation. VT100 or compatible terminals can be used, but some of the expanded features supported on the VT240 will not be available. For example, the menus will scroll up from one to the next instead of paging, lines requiring user entry will not be highlighted on the VDT, and plots will not be high-resolution.

3.3.2 IBM PC/XT or PC/AT Computers

The VSPS is also supported by Zoran under MS-DOS versions 2.1 and 3.1 for use on an IBM PC/XT or PC/AT personal computer with 640K bytes of memory. The installation procedure and directory structure recommended for the PC are presented in Appendix A.3.

Certain differences exist between the PC version of the VSP simulator and the version running on the VAX. These differences are described below:

Specification	MS-DOS version	VAX versions
VSP program and data	16K words total	64K words each
Maximum IEEE FFT size	1K points	8K points
Number of simulated VSPs	2	8
Application Library	Not in this release	Included

3.4 How to Use the VSPS

Use of the VSP simulator and its included utilities is entirely menu driven. Entering the simulator brings up the *Main* Menu from which all other menus can be reached either directly or indirectly. Instruction execution, waveform generation, plotting utilities, IEEE signal processing options, and the applications library are all examples of functions which may be invoked using this menu structure.

Instruction execution within the VSPS can be performed in one of two ways. The first of these is *interactively* using menus Individual instructions or sequences of instructions can be executed one-at-a-time. Because of the interactive nature of the menu structure, it is possible to execute an instruction, plot its output, generate a new waveform, have another instruction operate on this waveform, etc. This is a very powerful way of learning the VSP instruction set and associated parameters.

The second (and more powerful) way of executing VSP instructions is to write a program, or sequence of VSP instructions. The VSP simulator reads these instructions and models their hardware execution. This execution environment most accurately simulates the operation of the VSP system processor. Instruction files are created by using the host computer text editor. However, instruction display, waveform generation and terminal plotting utilities are all available interactively during program execution.

Chapter V contains a description of the macro command-language feature provided by the simulator. The macro command-language allows multiple keystroke simulator commands to be executed without using the simulator menus. This is a powerful and time-saving feature to use once experience has been gained with the VSPS menu structure. The macro command-language environment is called directly from the VSPS *Main* Menu and many of the subordinate menus. It is easy to get back to the menu environment from the command language environment.

3.4.1 Interactive Instruction Execution

Interactive instruction execution is designed as a menu-driven tutorial environment providing on-line interaction with the signal generator, individual VSP instructions and the waveform plotting utilities. Instructions are selected from a menu which displays the VSP instruction set. Parameter values corresponding to the selected instruction are also chosen interactively. The instruction can then be executed from within the same menu.

After each instruction is executed, VSP internal and external memory contents may be displayed and/or plotted on the terminal showing the results of the execution. This mode is especially useful for learning how the VSP instruction set and the associated parameters operate. It is also useful for experimenting with various instruction possibilities during algorithm development. The VSPS also has the ability to display the appropriate internal and/or external memory locations accessed on each VSP clock cycle during instruction execution.

The VSPS contains a signal generator for interactively creating either simple or sophisticated test signals. Signals which may be generated include sinusoids, square waves, step functions, impulse functions, and sums or products of the above. These waveforms can be stored in simulated external VSP memory or written to a disk file for later use or archival purposes in a waveform library. They can be plotted on the terminal both prior to and after instruction execution. Waveforms created from within other environments or programs by a user may also be displayed and used by the simulator in the same manner as waveforms created using the signal generator.

3.4.2 Programmed Instruction Execution

The VSP simulator has an operating environment which allows execution of full-length VSP source programs. An instruction file containing the VSP instructions and parameters is created with the system text editor and used as the source for execution. Breakpoints may be set within the source program at a selected clock count to allow examination of various memory or register contents during execution. The source may be single-stepped if it is desired to interrogate the RAM or registers during (or after) instruction execution. The simulator will keep track of the number of clock cycles taken to execute the program. This is especially important for calculations involving real-time operations.

One of the most powerful features provided by the simulator while running full-length VSP programs is the ability to model and simulate *the host system as well as the VSP*. Within a high-level 'C' language environment, the user can model the operation of the host system, or the environment external to the VSP, as well as the VSP itself. The host simulation will usually be written in 'C', while the VSP instructions are embedded within the 'C' program. The simulated external environment may include a user-created model of the system architecture, data and control buses, or any other desired component within the target system.

While the VSP code must be embedded within a 'C' program, it is not necessary to simulate the complete external environment in 'C'. For users more familiar or more comfortable with other high-level languages such as Fortran, it is possible to call Fortran subroutines from the 'C' program. In this case the 'C' program will contain only the VSP instructions and calls to Fortran subroutines. The Fortran subroutines will simulate the remainder of the target system. Full treatment on including Fortran subroutine calls in the 'C' simulation is included in chapter IX.

3.5 System Development Using the VSPS

This section discusses iterative algorithm development using the VSP simulator. In addition, it introduces the tools provided by the simulator for algorithm development and system simulation.

A block diagram of the normal algorithm development cycle using the VSPS is shown in Figure 3-1. It is expected that the user will first become familiar with the VSPS environment using the interactive signal generator, instruction tutorial and plotting capabilities.

The second phase of system development should be conceptualization of the algorithms to be executed by the VSP, followed by implementation of these algorithms using the VSP language. The process of algorithm definition and implementation using the VSP language is usually an iterative process. Once the algorithms are fully defined and implemented in VSP language, it is possible to simulate, verify and debug them completely within the simulator environment.

As the algorithms become more refined, it is possible to also include a high-level language simulation of the target architecture within the VSPS environment. At this point, the entire target system including the VSP, signal processing algorithms, and system architecture is modeled within the VSP simulator. Bus utilization, timing, arithmetic accuracy and throughput are all modeled for the complete system without the need to build the first prototype in hardware.

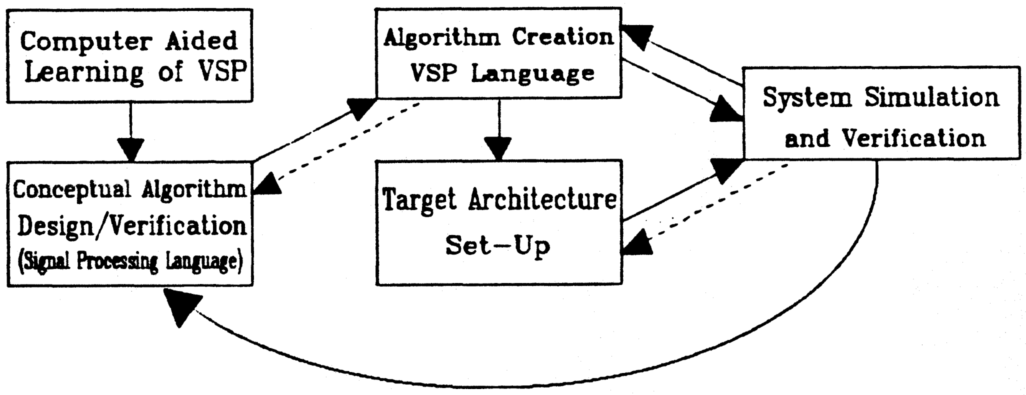

Computer Aided Learning of VSP
Algorithm Creation VSP Language
System Simulation and Verification
Conceptual Algorithm Design/Verification (Signal Processing Language)
Target Architecture Set-Up

Figure 3-1. The System Development Cycle Using the VSP Simulator.

3.6 The VSP Toolkit

The VSPS environment is a sophisticated environment for modeling both the VSP and complete target systems. Included within this environment is a powerful set of tools designed to simplify the task of system development. Figure 3-2 shows a block diagram depicting the order in which these tools are normally used in creating full-length VSP programs, along with target system simulation. Figure 3-2 does not apply to *interactive instruction execution*, which is a self-contained menu-driven instruction execution environment.

Once the algorithms are defined, a source program is created with a text editor resident on the computer system. If the program is simulating only the VSP, the 'C' source file contains only VSP instructions plus some VSP language constructs. If the source file is simulating a target system as well as the VSP, additional 'C' code is present which simulates the remainder of the target system (or calls other high-level language subroutines). In this case, the native VSP code is embedded within the high-level language simulating the host architecture.

After the source program is generated, it is run through the VSPS parser. The parser looks for VSP constructs that identify the VSP instructions which follow throughout the source file. At locations where VSP code exists, the parser translates the VSP instructions to external 'C' function calls to the VSP instruction library. If a target architecture is also being simulated in 'C', the standard 'C' code simulating the target system (or external subroutine calls) is not changed. Once the parsing is complete, the resultant 'C' program is compiled using the host computer 'C' compiler.

It should be noted that the parser always generates a 'C' program for compilation at its output. In cases where other simulation languages such as Fortran are desired, it is still necessary to generate a source program of VSP instructions within a 'C' subroutine for input to the parser. This is because the parser always requires a 'C' input file. However, the source program may be filled with calls to Fortran or other high-level language subroutines for simulating the remainder of the target system. It is not necessary to model the target system completely in 'C'. The parser will simply interpret the Fortran (or other high-level language) subroutine calls as an external function call. This eases the task for system developers who either are not well-versed in 'C' or who already have large bodies of existing software in another language. More detailed treatment is given to this subject in Chapter IX.

Once the program has been compiled, it must be linked with the main body of the simulator, VSP instruction library, applications library and the floating-point simulation utilities. At this point it may be executed, debugged and analyzed.

The VSPS includes a number of instruction queueing and timing models for instruction fetch, software fetch queues, and external bus access and timing. In the simplest case, the VSPS assumes no host controller operating system or on-device instruction FIFO. The VSPS will execute instructions sequentially in the source program until a "Halt" instruction is reached.

For applications requiring more accurate simulations of the target system bus priorities, usage, and accessing, as well as instruction fetch modeling, the VSPS includes extended instruction queueing and bus timing models. These extended models and the host/VSP timing coordination are covered in Chapter X.

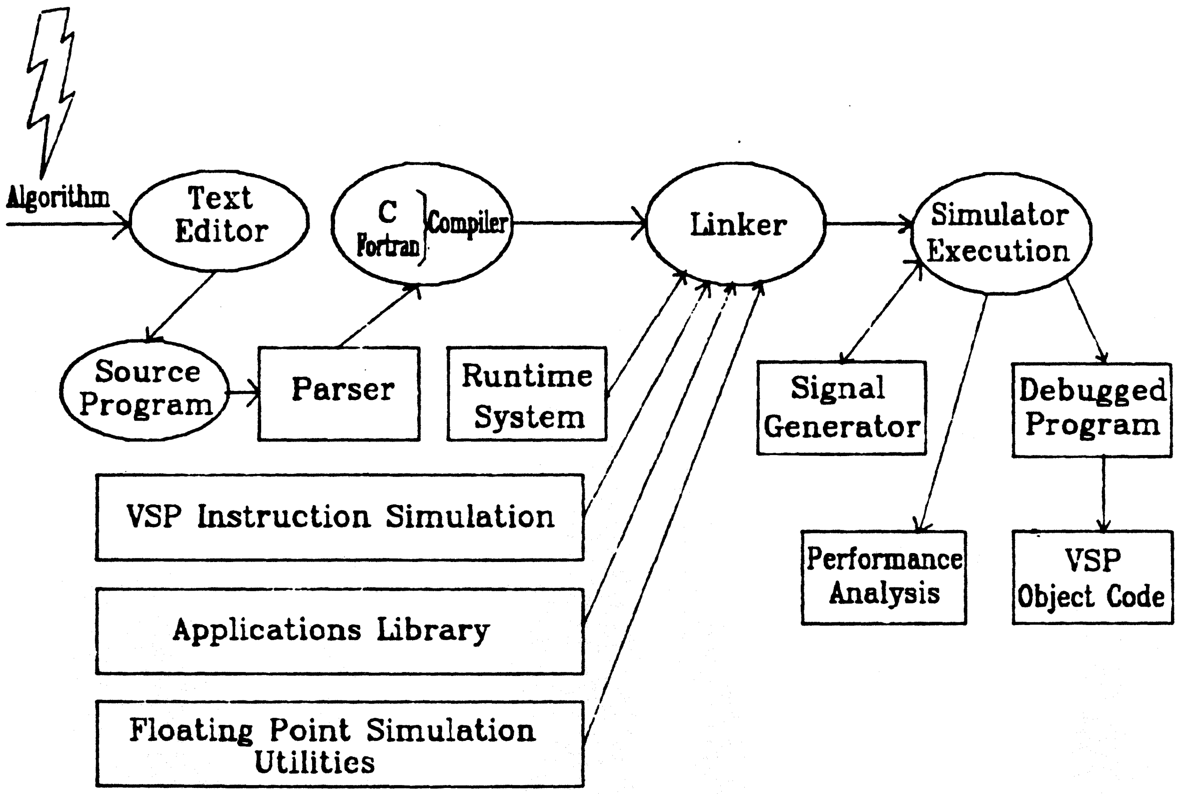


Figure 3-2. Software "Tools" Contained within the VSP Environment.

CHAPTER IV

VSPS INTERACTIVE USAGE

4.1 Overview

The VSP simulator operates in an *interactive* fashion. "Interactive" in this context means that the user is always guided to the next course of action by on-screen menus. The next action is always selected by typing a number which simply corresponds to the menu choices that appear on the screen. Included within the menu structure are HELP sections, tutorials on all of the VSP instructions, test data generation and display, selected IEEE signal processing options, and a library of pre-written application programs, as well as additional features for both sophisticated and novice simulator users.

This section will discuss all of the menus which appear throughout the simulator. Major section headings throughout the chapter correspond to the menu selections which appear on the terminal within the main simulator menu. Sub-menus will be discussed as sub-sections under these primary menus.

4.2 Aids to Learning

The VSPS incorporates "aids to learning" throughout the interactive menu structure, defined as HELP and TUTORIAL sections. These aids may be used when there is a question about the function of a menu or the options that the menu presents. It is a useful exercise when first learning the simulator and its menu organization to jump from one menu to the next and read the HELP and TUTORIAL sections. Using these selections will help the user master the simulator more quickly. Additionally, the simulator supports a *non-destructive* program interrupt at any point during program execution by typing a 'ctl-C'. This help facility is discussed in detail in section 4.4.8.

4.3 Conventions

There are five conventions to be understood which apply to the VSPS simulation of VSP instructions. They apply not only in this chapter, but also in the chapters to follow which explain the instruction set and the macro command-language.

4.3.1 Convention 1 - Literal and Logical Values

The first convention to be understood is the selection of values for instruction parameters as either *logical* or *literal*. The logical value is the *intuitive decimal value* assigned to the parameter. "Intuitive" means values which make the most sense to the user, not necessarily to the VSP. The logical value is translated by the simulator into a *literal value* to fill the field of the parameter in the instruction format. The *literal value* is the binary executable value used by the VSP. *The logical value is sometimes, but not always, equal to the literal value.*

The simulator user may describe instruction parameters to the simulator as either logical or literal values. Logical values are provided as parameter descriptors because they are oftentimes easier to understand and remember than are literal values. However, the literal values are the actual values which are stored in binary format in executable VSP code. When values are assigned to parameters, the "=" (equal sign) should be used for describing literal values, and the ":" (colon) for logical values. Literal and logical values are assigned interactively in Menu M-2-InOp. Translation formulas are also provided in this menu for instruction parameters which have logical values that differ from literal values. As a rule of thumb, it is usually easier to work with logical values than literal values.

For example, if the LD (LOAD) instruction is to be executed, one of the parameters is MBS (memory block size). Ignoring the definition of the MBS parameter, the *logical value* of the parameter may be 64 in one application. The *literal value* of the parameter is the $\log_2$ of the logical value (=6). However, the logical value is more descriptive of the parameter than the literal value. Other parameter values within instructions may also have different mathematical translations from their associated logical values. Menu M-2 allows the display of parameters in either logical or literal formats, as well as allowing switching between the logical and literal parameter representations.

4.3.2 Convention 2 - Internal/External Data Representation

The second convention for the VSPS is that when data in external RAM is used by the VSP, or moved to internal VSP RAM, the simulator shows the data as multiplied by a factor of two. This occurs because the 16-bit word in external memory is mapped into 17 bits internally by appending an extra LSB. This is equivalent to shifting the external word left by one bit when it is read into the VSP. When data is moved from the VSP internal memory out to external RAM, the data is divided by two; an effective shift right is performed to get the data back to 16 bits. When the contents of internal memory are displayed, the full 17 bits are represented. When external memory is displayed, the integer numbers represent 16-bit data words.

Note that internal RAM is actually 19 bits in length. The upper two bits are sign-extended for the purpose of overflow on intermediate results. They are not accessible externally, but are displayable from the simulator when an overflow occurs above the lower 17 bits in internal memory. It is assumed that the user scales the final results to fit in the lower 17 bits of internal memory.

4.3.3 Convention 3 - Full-Scale Integer Representation

The VSPS simulates the 16-bit word length contained in the VSP chip. These numbers are represented as 16-bit twos-complement integers in the simulator. The maximum positive number which can be represented is 2^{15}-1 (32767). The smallest negative number represented is -2^{15} (-32768). The floating-point number +1.0 is represented by 32767/32768, and -1.0 is represented by -32768/32768. Note that +1.0 cannot be represented exactly in the twos-complement number system.

The sine/cosine look-up tables provide an extra bit of precision, providing a minimum integer of -65536 and a maximum integer value of 65536. Special logic is provided to allow both +1.0 and -1.0 integer representations out of the look-up table.

4.3.4 Convention 4 - Internal and External Memory Addressing

The fourth convention has to do with the way data is stored in external RAM and in internal VSP RAM. For purposes of this discussion, *external RAM* refers to simulated RAM external to the VSP chip, but as seen and addressed by the VSP simulator. The VSPS uses references to the *index value* of data points in external RAM in some cases, and to the physical address of these data points in external memory in others. This will be elaborated upon in the following paragraphs.

4.3.4.1 Internal Memory Addressing

The VSPS refers to the "jth" point in *internal* VSP RAM as follows:

jth real point (R_j):	VSPRAM [j]
jth imaginary point (I_j):	VSPRAM [j.1]
jth complex point (R_j,I_j):	VSPRAM [j]

where:

j is in the range 0 to 127,
R_j = real part,
and I_j = imaginary part.

The above description of internal VSP RAM also implies that while the VSP contains 256 words of internal RAM, it is partitioned into 128 complex words addressable from 0 to 127. Real data always exists in the real RAM section, while imaginary data always exists in the imaginary section. It is not possible to re-partition this RAM into one consecutive section of 256 real samples or 256 imaginary samples. Real-only and imaginary-only data is allowed within the VSP, but these vectors are limited in length to 128 points. Later discussions will show that it is possible to perform either real or imaginary operations (in addition to complex operations), but the partitioning of internal RAM is always as 128 complex words.

Internal VSP RAM can store 128 real, imaginary or complex data samples. It is useful to think of this RAM as an array of 128 words of 38 bits; the first 19 bits being the real component and the next 19 bits being the imaginary component. This is useful because of the way the VSP indexes into this array. The jth index always addresses the jth complex word.

4.3.4.2 External Memory Addressing

Simulated data memory external to the VSP is treated differently than is internal VSP RAM. The VSP has a 16-bit external address bus which limits the direct physical address space of the VSP to 64K words. 64K real samples, 64K imaginary samples or 32K complex samples can be stored in the available 64K x 16-bit external address space. Real-only data samples or imaginary-only data samples may be defined within the simulator to occupy successive 16-bit physical address locations in memory (64K 16-bit words). Complex data samples are defined as alternating successive physical addresses for the real and imaginary parts of one complex word. The real component of the complex word is stored in the even address, while the imaginary component is stored in the odd address. Hence, 32K complex (32-bit) words will fit into the 64K (16-bit) word simulated address space of the VSP.

In terms of VSP simulator interpretation of this nomenclature, *the VSPS will ordinarily display simulated external data memory as an array of 32K complex samples.* It is important to think of the *complex indices* of the samples being referenced in external memory and not the physical addresses of these samples. When the simulator prompts for an *address*, it is referring to a simulated physical 16-bit address in external memory. When the simulator prompts for a *sample index*, it is referring to either a complex (32-bit) or real (16-bit) sample index depending on the context.

Using this description of simulated external RAM, the VSP views external memory in a similar manner to the way it has physically partitioned internal memory; as 32K words of complex samples. Hence, the indexing address space of the 32K complex samples is *half* the 64K physical address.

There is, however, one difference between an external memory reference and an internal VSP RAM reference. This is the fact that the VSP internal memory is physically partitioned into a *real* part and an *imaginary* part. The VSP has no direct control over the type of data which exists in external RAM. It may be real, imaginary, or complex, and occupy successive or alternate memory locations. Because of the variety of the data which may be stored in simulated external memory, the VSPS has adopted the nomenclature for referencing this external memory as a 32K x 32-bit complex array, independently of the type of data actually stored there.

The following index definition will illustrate how the VSPS addresses external memory. Assume that the full 64K external memory address space of the VSP is filled with data. If the data stored in external RAM is:

either real data OR *imaginary data*:

D_j = ExtRAM [j/2]
D_{j+1} = ExtRAM [(j/2).1]

complex data:

(R_j,I_j) = ExtRAM [j]
(R_{j+1},I_{j+1}) = ExtRAM [j.1]

where:

j is always even: 0, 2, 4, ... 65534
D is either a real OR imaginary data array,
R is the real portion of a complex data array,and
I is the imaginary portion of a complex data array.

When specifying a Memory Base Address (MBA) to the VSPS for external memory, a *physical address* is always used, not an index. This is the only exception to the above discussion of indexing into the complex external memory. Hence, there is occasionally a mental conversion required between *indices* and *physical memory addresses* when the memory base address is specified. The reason for this inconsistency is that the VSP system processor requires a physical address upon which to begin addressing external memory.

4.3.5 Convention 5 - Display of Internal VSP Memory

When the simulator displays the contents of the internal VSP RAM, it is referenced as:

.
.
VSPRAM:*n*[j](r_j,i_j),
.
.

where:

n refers to the logical number of the VSP whose memory is being displayed (default 0),
j is the index of the complex array,
r_j is the real part of the *jth* complex word, and
i_j is the imaginary part of the *jth* complex word.

The convention as described above allows display of the contents of the internal RAM when *multiple* VSPs are being simulated in a single application by the VSPS.

4.4 Using the Menus

The VSP simulator is called from the operating system by typing 'vsps' on the VAX or 'vsps' on the PC followed by a carriage return <cr>. In the PC version, the first thing to appear on the screen will be a copyright notice. The first menu to appear on the screen is the *Main* Menu, the starting point for all interactive use.

A structure exists within the VSPS on-screen menus, as can be seen by the menu tree structure shown in the reference card accompanying this manual. At the top of the structure is the *Main* Menu, leading to all subordinate menus and options existing below the *Main* Menu. Regardless of which menu path is traversed by the user, it is always possible to get back to either the previous menu or the *Main* Menu directly. Each lower-level menu has either an optional or an automatic return to the next highest menu level.

Most of the prompts at the end of a menu present two numbers in parentheses. The first number returns operation to the previous menu level to correct an entry. The second is the default selection, which is executed if a <cr> is entered without a value.

The menus and selections are identified in this manual by codes which simplify understanding the menu structure and to get directly to a subordinate level of choice. The *Main* Menu is defined as "M" (for *main*). Each subordinate menu or choice is named with "M-" followed by the number of the menu. The second level of menus will have another hyphen and another number identifying the choice. Observing the menu naming structure shown in the menu reference card should make this naming convention clear.

Each prompt line will be followed by one or two options. The first option gives the user the opportunity to return to the previous line. It is very useful for correcting errors made in entries. The second option is the default value, which will be automatically selected if the user does not specify his own value. In some cases only one option - the default value - will appear. If there is only one option presented, it indicates that the user may not return to the previous line.

4.4.1 The Main Menu - Menu M

Figure 4-1 shows what the *Main* Menu looks like on the user screen. This is the menu arrived at upon calling the VSP simulator. The first choice available in the *Main* Menu is to select the HELP screen (M-1). The HELP screen discusses the philosophy of the VSPS and suggests how to get started. It is worthwhile to execute the HELP command from the *Main* Menu before proceeding further.

Menu selections M-2 through M-7 are subordinate menus, and are described in the following sections.

Menu selections M-8, M-9 and M-10 provide slighty more sophisticated (and powerful) features of the simulator. Menus M-8 and M-9 are used to execute and debug user programs. Menu M-10 enters the *macro command-language* environment of the simulator. Selection M-11 halts execution of the VSPS program and returns control to the operating system.

```
MAIN MENU

1 HELP
2 VSP instruction tutorial and execution
3 Data generation and display
4 Display options, timing control and queueing
5 Signal processing library for VSP
6 IEEE signal processing library
7 Application library
8 Execute user program in vspop()
9 Execute batch commands and VSPS validation
10 Command mode
11 Exit

Specify value of your selection (0)(1):
```

Figure 4-1. Main Menu - Menu M.

Default selection 1 will call the HELP section. The default option may be selected by either typing a <cr> with no parameters or a '1' followed by a <cr> at the prompt.

4.4.2 Instruction Selection Tutorial Menu - Menu M-2

This menu is the main reference and tutorial on the instructions in the VSP language. It also allows interactive parameter specification as well as execution of VSP instructions. Figure 4-2 shows what Menu M-2 looks like on the user screen. Two classes of selections are provided by this menu.

The first selection contains a series of initialization operations along with a very useful HELP option. These are shown in the upper half of Figure 4-2, and are identified as menu options '1' through '12'. Selections 2, 3, 4, 6, 7 and 8 each initialize features in the VSPS which control later execution. Selection 5 dumps the descriptions of all instruction parameters to a disk file.

The second selection class in this menu is the complete list of VSP instructions, shown in the lower half of Figure 4-2. When the desired instruction name is entered at the prompt, a menu appears with information for that instruction: format, parameters, range of parameter values, constraints and other useful information. The following section, 4.4.2.2, discusses the usage of the menu in more detail. Chapter VI describes VSP instruction usage and parameter descriptions in great detail.

```
INSTRUCTION SELECTION TUTORIAL MENU (M-2)
Enter an instruction name or a numeric option.

1 HELP
2 Set CYCMEM in mode register
3 Set number of RAM sections in mode register
4 Initialize scale nibbles
5 Dump instruction parameter descriptions to file
6 Set the VSP number
Set bit numbering:  7 right to left   8 left to right
VSP interrupts:  9 enable   10 disable
11 Modify internal RAM
12 Return to previous menu

Instruction names are:
NOP        JMPI       LDSM       LD         STB
ST         STI        MLTC       MLTR       ADDC
ADDR       FFT        DEMO       MODLT      SCLT
SCL        ABS        CMCN       MGSQ       CMLT
ACCI       ACCR       HLT

Specify instruction or numeric option for demonstration (12):
```

Figure 4-2. Instruction Selection Tutorial Menu - Menu M-2.

Description of options for Menu M-2

This section describes the options which are available under the *Instruction Selection Tutorial* Menu (M-2):

1 HELP
Describes the *Instruction Selection Tutorial* Menu

2 Set CYCMEM in mode register
Allows interactive setting of the CYCMEM parameter contained in the VSP mode register. Chapter V discusses the function of CYCMEM in greater detail.

3 Set number of RAM sections in mode register
Allows interactive setting of the NMS bit contained in the VSP mode register. Chapter V discusses the function of NMS in greater detail.

4 Initialize scale nibbles
Allows interactive setting of each of the scale nibbles contained in the internal VSP scale RAM. Each nibble should be entered as a *decimal number*, not a hex number, from 0 - 15.

5 Dump instruction parameter descriptions to file
This option dumps a description of all of the parameter fields for all of the instructions to a disk file. Included are parameter defaults, widths, logical and literal values. The file may be printed for quick reference on the VSP instruction parameters.

6 Set the VSP number
Allows defining the logical number of the current VSP to a number other than the default of 0. This is necessary for multiple VSP applications.

7 Bit numbering right-to-left
Allows redefining the bit numbers in the instruction fields to bit number 0 on the right end of the instruction word and to bit number 15 on the left end of the word. This is the default format of instruction words, and is compatible with standard nomenclature.

8 Bit numbering left-to-right
Allows redefining the bit numbers in the instruction fields to bit number 15 on the right end of the instruction word and to bit number 0 on the left end of the word.

9 Enable VSP interrupts
Sets the four LSBs of the mode register to 1. Each bit enables a particular interrupt condition for the VSP.

10 Disable VSP interrupts
Clears the four LSBs of the mode register to 0. The VSP will not generate an interrupt as a function of the four LSBs in the mode register.

11 Modify external RAM

Shows memory locations one by one, displaying the current value and allowing the user to change those values as necessary.

12 Return to previous MENU

Exits the *Instruction Selection Tutorial* Menu and returns control back to the *Main* Menu.

Instruction Name

Typing any instruction name (shown in the lower half of figure 4-2) at the prompt in Menu M-2 calls Menu M-2-InOp. This menu allows instruction parameter definition as well as instruction execution. Section 4.4.2.1 discusses this in more detail.

4.4.2.1 Instruction Options Menu - Menu M-2-InOp

The *Instruction Options* Menu (M-2-InOp) is entered by selecting an instruction from the lower half of Menu M-2. An example of Menu M-2-InOp is shown in Figure 4-3. The upper half of the menu displays the default parameters defined by the system for the particular instruction chosen. Additionally, a menu of choices for additional information about the instruction is displayed in the bottom half of Menu M-2-InOp. The lower half of the menu is the same for all instructions, although the displayed parameters in the upper half are particular to the instruction chosen.

```
Instruction parameters (with LOGICAL values) for LD are:

NMPT :64    RS   :0    MDF  :3     INTRP      :0
ZP   :0     EI   :0    MBS  :128   MSS        :2
RV   :0     ZR   :0    AD   :0     MBA        :0

0 executes instruction 'LD'. (If the message level is 1 or higher, the
instruction and its parameters will be displayed. If the message level is 2
or higher, the display will be followed by a read to the terminal. At this
point, you may enter a new message level or any Menu M-4 option
followed by a <cr>. If you set the message level to 3, a detailed
description of instruction execution will be displayed.)

1 display size and restrictions of all parameters
2 display size and restrictions of all parameters and opcodes
3 display and set LOGICAL parameter values
4 display and set LITERAL parameter values
5 show the translations from LOGICAL to LITERAL values
6 return to Instruction Selection Tutorial Menu

Specify instruction parameter to change or numeric option above (6):
```

Figure 4-3. Instruction Options Menu - Menu M-2-InOp.

Description of menu choices for Menu M-2-InOp:

1 Display size and restrictions of all parameters
Displays the number of bits required of each parameter in the instruction word, the first bit location, the word number for instructions that require multiple words, and the defaults for the parameters.

2 Display size and restrictions of all parameters and operation codes
Same as display for option 1 plus additional parameters

3 Display and set LOGICAL parameter values
Displays all instruction parameters in the upper half of Menu M-2-InOp as LOGICAL values. All logical values are preceded by a colon.

4 Display and set LITERAL parameter values
Displays all instruction parameters in the upper half of Menu M-2-InOp as LITERAL values. All literal values are preceded by an equal sign.

5 Show the translations from LOGICAL to LITERAL values
Describes which parameters have different LOGICAL and LITERAL descriptions. The translation formula is given for these parameters.

6 Return to Instruction Menu
Returns from the Instruction Parameter Menu (Menu M-2-InOp) to the Instruction Menu (M-2).

Within Menu M-2-InOp, after the instruction is displayed and before it is executed, the user may use any of the Menu M-4 options by simply typing in the command number corresponding to the function desired (see Menu M-4, page 4-18).

Figure 4-3 shows a specific example of the screen created when the LD (Load) instruction is entered from menu M-2. The upper half of the screen displays the default LD parameters as they are defined by the system. The lower half of the menu allows display of certain options and setting of logical and literal parameter values. The instruction parameters in the upper half of Figure 4-3 are unique to the LD instruction. Other instructions would have a similar display, but with different parameters. The six numbered menu choices in the lower half of the figure are common to all instruction displays.

Any of the instruction parameters shown in the upper half of the menu may be easily changed by simply typing the parameter name followed by a <cr> at the prompt in Menu M-2-InOp. A prompt will be displayed with a query on the new value to be assigned. For example, if the number of points to load is desired to be something other than what is displayed, typing 'nmpt' <cr> will prompt the user for the new number of points to load. The correct response is any integer less than or equal to 128. As new parameters are entered interactively, the instruction parameters in the upper half of the menu are updated. Typing a <cr> (default 0) at the prompt will clear Menu M-2-InOp and replace the display with the instruction selected and its list of associated parameters. Typing a second <cr> will execute the instruction with the displayed parameters.

Just prior to the second <cr> which begins instruction execution, the user has the option to change the message display level for the instruction during execution. Different display levels provide different amounts of information about the instruction on the screen while the instruction is executing. Table 4-1 shows the four different message level capabilities of the simulator. The default level is '2' when the simulator is entered. This level will display the instruction to be executed, but not the memory contents addressed on each clock cycle. It is used primarily for single-stepping instruction execution. For the purpose of this discussion, type a '3' prior to the second <cr>. Both internal VSP RAM as well as external RAM being addressed on each clock cycle is displayed.

Table 4-1. Display levels provided during instruction execution.

0	print nothing
1	print VSP instructions
2	print VSP instructions and wait for RETURN or selection from this menu
3	print detailed description of each instruction

The message level of the simulator may also be changed in Menu M-4. In fact, the options shown in Table 4-1 are taken directly from the option of Menu M-4. Section 4.4.5 will discuss Menu M-4 in greater detail.

If message level 2 or 3 is selected from Table 4-1, it will apply to all later interactive instruction executions until it is manually changed by entering another option. If message level 0 or 1 is selected, the message level may not be changed from Menu M-2-InOp. The message level must be changed by going through Menu M-4.

NOTE: It is recommended that the VSP engineering data sheet be consulted for detailed descriptions of all parameters associated with the instruction set.

4.4.3 Data Generation and Display Menu - Menu M-3

Figure 4-4 shows a picture of the screen created by the *Data Generation and Display* Menu (M-3). From this menu options can be chosen which assist in creating input signals, writing output signals to memory, initializing memory, dumping VSP memory, and plotting data in external memory. Option '15' is used to choose the seed for the random number generator. The HELP selection is very useful while becoming familiar with these menu selections. The only option which leads to a subordinate menu is option '8', which calls the interactive signal generator.

```
DATA GENERATION AND DISPLAY MENU (M-3)

1 HELP
2 Command mode
3 Previous menu


External Memory:                VSP Internal Memory:
 4 Modify memory                9 Modify memory
 5 Graphics plot                10 Graphics plot
 6 Character plot               11 Character plot
 7 List                         12 List
 8 Signal generation            13 Display Registers
                                14 Select VSP


Other Options
 15 Set random number generator seed
 16 Initialize all memory

Specify value of your selection (0)(3):
```

Figure 4-4. Data Generation and Display Menu - Menu M-3.

Description of Option Selections for Menu M-3:

1 HELP

Describes the *Data Generation and Display* Menu (M-3).

2 Command Mode

Selects the macro command-language environment. This is the same environment which can be entered from the *Main* Menu.

3 Previous Menu

Returns the simulator to the previous menu.

4 Modify memory
Allows interactive entry of data in external memory at specific addresses; data may be specified in either decimal or hexadecimal form.

5 Graphics Plot
Will use the graphics capabilities of the terminal for graphically displaying simulated external VSP memory. Options allow plotting of signals in the following formats: complex signals, real or imaginary parts of complex signals or packed real signals. *Packed real* signals indicate that successive external memory addresses are interpreted by the plotting routines as real (imaginary) signals. This contrasts with plotting the real or imaginary parts of a complex signal where every other external address is displayed.

In addition, the *difference* between two signals may also be plotted.

Users are interactively queried when selecting this option.

6 Character plot
Allows plotting simulated external VSP memory on display terminals which do not have graphics capability. The queries for this option are exactly the same as for option '5'.

7 List
This option dumps the contents of external memory locations selected by the user. The queries for this option are exactly the same as for options 5 and 6.

8 Signal Generation
Calls the simulator *Signal Generation* Menu (M-3-8). Menu M-3-8 is able to generate a number of different kinds of signals which may be used for input to the simulator. As signals are generated, they are written to simulated external VSP memory. Menu M-3-8 is discussed in detail in section 4.4.3.1.

9 Modify memory
Allows interactive entry of internal data in specific location.

10 Graphics plot
Will use the graphics capabilities of the terminal for graphically displaying simulated internal VSP memory. Options allow plotting of signals in the following formats: complex signals, real or imaginary parts of complex signals or packed real signals. *Packed real* signals indicate that successive internal memory addresses are interpreted by the plotting routines as real (imaginary) signals. This contrasts with plotting the real or imaginary parts of a complex signal where every other internal address is displayed.

In addition, the *difference* between two signals may also be plotted.

Users are interactively queried when selecting this option.

11 Character plot

Allows plotting simulated internal VSP memory on display terminals which do not have graphics capability. The queries for this option are exactly the same as for option '10'.

12 List

This option dumps the contents of internal memory locations selected by the user. The queries for this option are exactly the same as for options 10 and 11.

13 Display registers

Displays contents of VSP accumulators, scale registers, mode and status registers.

14 Select VSP

Allows selection of the logical number of the VSP. This option is used only when multiple VSPs are being simulated in a single application within the VSPS. The default is '0'.

15 Set random number generator seed

The user can set the seed for the random number generator. The seed for the random number generator need not be set, but different seeds for successive runs will ensure different random number sequences. Random number sequences are generated in the *Signal Generation* Menu (M-3-8).

16 Initialize all memory

This option initializes all simulated external memory as well as internal VSP memory associated with the simulator to non-zero values. The values written to memory are a ramp function beginning at data value 16384.

4.4.3.1 Signal Generation - Menu M-3-8

The VSP simulator contains a signal generator that creates various types of test signals which are useful for testing algorithms. The signal generator is entered from the *Data Generation and Display* Menu (M-3) as option '8'. Figure 4-5 shows what the *Signal Generation* Menu looks like on the terminal screen.

```
SIGNAL GENERATION MENU (M-3-8)

1 HELP
2 Signal generation with normalization after summation
3 Clear memory and set parameters for a new signal
4 Add a signal to memory
5 Subtract a signal from memory
6 Multiply memory by a signal
7 Divide memory by a signal
8 Save signal to disk as integers
9 Load integer signal from disk
10 Save a signal to a hex format file
11 Load a signal from a hex format file
12 Command mode
13 Return to previous menu

Specify value of your selection (0)(13):
```

Figure 4-5. Signal Generation Menu - Menu M-3-8.

Description of menu choices for Menu M-3-8:

1 HELP

Describes the *Signal Generation* Menu and its options.

2 Signal generalization with normalization after summation

Generates a compound signal. A compound signal is a signal which may be a summation of a number of signals such as sine waves, square waves, exponentials, etc. All arithmetic is done in floating-point and then converted to integer **after** the complete signal is generated. This particular signal generator is a complete package and can be used in place of options 3 through 11, which are discussed later. In general, this signal generator OR the integer signal generator made up of options 3 through 11 should be used.

The user is queried about the following:

A) the **memory base address** (physical address in simulated external memory) to which the data should be written;

B) **total amplitude** of the signal, to which the appropriate response is any real number between -1.0 and +1.0. This fraction is scaled to a 16-bit twos-complement full-scale integer value. Negative numbers specify that the data generated is real. Positive numbers specify that the data generated is complex;

C) **number of samples generated;**

D) **bit-reversed addresses.** The user can specify
'1' for bit-reversal, or '2' for all 0 data.
The default value is '0' and serves to delineate the data in normal order.

In addition to the "sum of sines" option, the user is presented with two other options. The first is a "special" function which allows the user to perform a "sum of sines" with exponentially increasing amplitude. (This option, although rarely used, is primarily for the purpose of debugging FFTs whose actual value may be somewhat masked by the symmetry of the signals generated.) The second is a "save" function which enables the user to restore data saved in either one of the memory save buffers or on disk.

Following signal generation, the user is queried as to where the signal is to be stored. Values 1-10 represent memory save buffers, in which the signals are stored as non-scaled floating-point. Values 11-13 will save the signal to a disk file as follows:

11	Single precision floating-point
12	16-bit Integer
13	Double precision floating-point

3 Clear memory and set parameters for a new signal

Clears an amount of simulated external VSP memory defined by the user. This is usually done prior to generating a new signal. The user is queried as to the physical starting address of memory to clear, the number of samples, and whether the data is real or complex.

4 Add a signal to memory

Adds a new signal to simulated external VSP memory. The result is stored in simulated external VSP memory. Types of signals which may be added are: sine, cosine, and square waves, random signals, DC values, impulses and ramps. This function may be used in conjunction with options 5 through 7 to create compound signals which are formed as sums or products of other signals already existing in memory.

In this selection, the user is queried about several options after selecting the type of signal to be generated. The user is queried about the following:

A) the **relative amplitude** of the signal. This refers to the amplitude of the new signal with respect to the signal previously generated. This is very important as it determines how the new signal will be scaled in respect to the original signal.

B) the **phase in degrees** of the initial phase of the signal to be generated.

C) the **cycles per total samples.** This refers to the number of cycles in the total number (already specified by the user) of samples, and is independent of the sample rate.

5 Subtract a signal from memory

Performs the same function as option '4' except that a signal is subtracted from simulated external VSP memory.

6 Multiply memory by a signal

Performs the same function as option '4' except that a new signal is multiplied with a signal which exists in simulated external VSP memory.

7 Divide memory by a signal

Performs the same function as option '4' except that a signal existing in simulated external VSP memory is divided by a new signal.

8 Save signal to disk as integers

Allows saving a signal which exists in simulated external VSP memory to disk. This is useful for archival purposes or for building a library of signals. The file is saved in the same binary format as it is generated in, with the first four characters serving as a counter.

9 Load integer signal from disk

Allows reading a signal which exists in a disk file (in binary format) into simulated external VSP memory.

10 Save a signal to a hex format file

Allows the user to save a signal which exists in simulated external VSP memory to a hex-format file. In this case, the signal is saved in *Intel Hex* form. The first and last lines serve as the header and trailer, respectively. The first four characters of each line refer to the line number, the second four represent the actual number of bytes, and the final four serve as a check sum. This check sum allows the user to verify that the signal was transmitted without error from the original binary format to the hex-format.

11 Load a signal from a hex format file

Allows reading a signal which exists in a hex-format file (see option '10', above) into simulated external VSP memory.

12 Command mode

Selects the macro command-language environment. This is the same environment which can be entered from the *Main* Menu.

13 Return to the previous menu

Returns the simulator to the previous menu.

The difference in using option '2' versus options 3 through 11 for signal generation is that option '2' performs all arithmetic in *floating-point* and converts the results to integers after the complete signal is generated. Options 3 through 11 perform all signal generation using the 16-bit integer representation of simulated external memory.

4.4.4 Clock Control and Other Display Options Menu - Menu M-4

Figure 4-6 shows what Menu M-4 looks like on the user screen. This menu provides a variety of choices, and is useful in helping debug algorithms as well as displaying or plotting results. The upper half of the menu (options 0 - 3) is used to choose the amount and kind of message-level display desired on the terminal while instructions are being executed. The lower half provides utility functions for displaying the state of VSP instructions as they are executed: plotting data, displaying memory and registers, displaying source code, and displaying data on an individual VSP when several are in use. Three of the options -- 20, 21 and 22 -- lead to subordinate menus. These subordinate menus will be discussed in sections 4.4.4.1, 4.4.4.2, and 4.4.4.3.

```
DISPLAY OPTIONS MENU (M-4)

Message levels are:
0 print nothing          1 print VSP instructions
2 print VSP instructions and wait for
  RETURN or selection from Display Options Menu
3 print detailed description of each instruction
9 HELP

Display external memory:
10 Graphics plot         11 Character plot        12 List

Display VSP internal memory:
13 Graphics plot         14 Character plot        15 List
16 Display registers     17 Select VSP

Display source code file and line number with instructions:
18 Start                 19 Stop

Other menus:
20 Additional display options
21 VSP timing control and display
22 Control of instruction queueing and breakpoints

Setting the message level exits from this menu

Specify value of message level or other option (-1)(2):
```

Figure 4-6. Display Options Menu - Menu M-4.

Note while executing user or library programs at message levels 2 or 3, it is always possible to immediately execute any of the options shown in this menu, *even when the menu is not displayed.* When message level 2 or 3 is being used during program execution, each instruction requires a <cr> prior to its execution. If, instead of typing a <cr>, a user types any of the options displayed in Menu M-4 followed by a <cr>, the option chosen will be executed. For instance, typing '10' <cr> will immediately begin querying the user about high-resolution plotting parameters. Also, typing a '1' <cr> will change the message level to 1 so that instructions are printed to the screen when they are executed, but they do not require a carriage return before they are executed.

Description of Menu Choices for Menu M-4:

0 Print nothing

No messages are printed to the terminal regarding the execution of VSP instructions. After each instruction is executed, control is returned to the calling menu.

1 Print VSP instructions

The VSP instruction being executed is printed on the screen as it is executed. No additional information about the instruction is printed.

2 Print VSP instructions and wait for RETURN or selection from this menu

Each VSP instruction is printed on the terminal prior to its execution. Before it can be executed, the user must confirm its execution by hitting a <cr> or providing a new help level (0-3) followed by a <cr>.

3 Print detailed description of each instruction

Each VSP instruction will be printed on the terminal prior to execution as in message level 2. Its execution must again be confirmed by a <cr>. After the <cr> the Simulator will display the contents of any internal or external memory (or both) being accessed by the instruction on each clock cycle.

9 HELP

Describes the *Display Options* Menu and its options.

10 Graphics plot

Will use the graphics capabilities of the terminal for graphically displaying simulated external VSP memory. Options allow plotting of signals in the following formats: complex signals, real or imaginary parts of complex signals or packed real signals. *Packed real* signals indicate that successive external memory addresses are interpreted by the plotting routines as real (imaginary) signals. This contrasts with plotting the real or imaginary parts of a complex signal where every other external address is displayed. The queries for this option are exactly the same as those for option '5' in Menu M-3.

In addition, the *difference* between two signals may also be plotted.

Users are interactively queried when selecting this option.

11 Character plot

Allows plotting simulated external VSP memory on display terminals which do not have graphics capability. The queries for this option are exactly the same as for option '10' above.

12 List

This option dumps the contents of external memory selected by the user. The queries for this option are exactly the same as for options '10' and '11' above.

13 Graphics plot

Will use the graphics capabilities of the terminal for graphically displaying simulated internal VSP memory. Options allow plotting of signals in the following formats: complex signals, real or imaginary parts of complex signals or packed real signals. *Packed real* signals indicate that successive internal memory addresses are interpreted by the plotting routines as real (imaginary) signals. This contrasts with plotting the real or imaginary parts of a complex signal where every other internal address is displayed. The queries for this option are exactly the same as for option '5' in Menu M-3.

In addition, the *difference* between two signals may also be plotted.

Users are interactively queried when selecting this option.

14 Character plot

Allows plotting simulated internal VSP memory on display terminals which do not have graphics capability. The queries for this option are exactly the same as for option '13' above.

15 List

This option dumps the contents of internal memory selected by the user. The queries for this option are exactly the same as for options 13 and 14 above.

16 Display registers

This option displays the accumulators, scale registers, mode and status registers of the simulated VSP with logical number 0 to the terminal. (This is the same display as provided in option '13' in Menu M-3.)

17 Select VSP

Performs the same dump as described in option '15' above. However, this option can dump the contents of VSPs which are not currently active. This option is valuable for applications simulating multiple VSPs.

18 Start display of source code file and line numbers

When VSP programs are executed, this option will display on the terminal the source code file name and line number for each VSP instruction executed. It is useful for tracking program execution. The option is defaulted ON when the simulator is run.

19 Stop display of source code file and line numbers

Stops printing on terminal of the source code file names and line numbers when VSP programs are executed with the VSPS. This option turns off option '18', above.

20 Additional display options

This option calls a subordinate menu which contains additional display options. The options available under the subordinate menu are discussed in section 4.4.4.1.

21 VSP timing control and display
This option calls a subordinate menu which initiates the VSP clock options. The options available under the subordinate menu are described in section 4.4.4.2.

22 Control of instruction queueing and breakpoints
This option selects a subordinate menu which controls the VSPS queueing options and allows setting a breakpoint on a particular clock cycle. The queueing options affect the simulation of instruction fetch and execution. The options available under the subordinate menu are described in section 4.4.4.3.

4.4.4.1 Additional Display Options Menu - Menu M-4-20

Figure 4-7 shows what the *Additional Display Options* Menu (Menu M-4-20) looks like on the terminal.

Option 1 is the HELP selection for Menu M-4-20.

Option 2 returns control to the calling menu (Menu M-4)

Options 3, 4, 5 and 6 are switches which select the data representation when memory dumps are performed.

Options 7, 8 and 9 control instruction and comment dumps to disk.

Option 10 turns on and option 11 turns off the display of the file name containing the HELP text prior to the simulator reading the HELP file. This is useful for instances where users may want to add additional textual information to existing HELP files. The default when the simulator is run is not to display HELP file names.

Options 12 and 13 change the display of instruction parameters between logical and literal formats. These last two are the same as options 3 and 4 in Menu M-2-InOp.

Options 14 and 15 control the formatted dumping of memory and instructions to disk.

```
ADDITIONAL DISPLAY OPTIONS MENU (M-4-20)

1 HELP
2 Previous menu

Memory and register display format:
3 Integer    4 Real   5 Hex    6 Binary

Disk dump of instructions and plots:
7 Start    8 Stop    9 Add comment line

Display of help file names when files are read:
10 Start          11 Stop

Instruction parameter display:
12 LOGICAL     13 LITERAL

Other:
14 JEDEC format instructions dumps
15 DAISY format dumps

Specify value of display option (0)(2):
```

Figure 4-7. Additional Display Options Menu - Menu M-4-20.

4.4.4.2 VSP Timing Control and Display Menu - Menu M-4-21

Figure 4-8 presents the clock options available in the VSPS environment for menu M-4-21. Included are choices for displaying clock operation, turning the clock on and off, clearing it, and setting the speed (in clock cycles) of external RAM. Instruction execution times can be determined by clearing the clock and having clock cycles displayed during instruction execution. By setting the VSPS to execute for N clock cycles, the program will stop after N cycles at which time the state of the VSP registers and memory may be interrogated. A HELP menu is provided as option '1' which explains how to use the options provided by the menu.

```
VSP TIMING CONTROL AND DISPLAY MENU (M-4-21)

1 HELP
2 Return to previous menu

Print clocks with instructions:
3 Start         4 Stop

Clock:
5 On            6 Off

Cumulative clock and bus cycles:
7 Clear         8 Display

Other:
9 Set external RAM speed
10 Set clock speed
11 Set break point at N clocks

Specify value of clock option (0)(2):
```

Figure 4-8. VSP Timing Control and Display Menu - Menu M-4-21.

4.4.4.3 Control of Instruction Queueing and Breakpoints Menu - Menu M-4-22

Figure 4-9 shows the options available in Menu M-4-22 for modeling VSP instruction queueing and related parameters. The upper half of the menu either turns off the modeling of the VSP hardware queue or determines when control is returned to the host after an instruction is written to the queue. The second half of the menu provides options for instruction queueing. A short on-screen explanation of the choices exists on the terminal under the menu heading. A more detailed treatment of host/VSP coordination and instruction queueing is presented in Chapter X.

Responding to Menu M-4-22 with any positive number sets a breakpoint at that number of clocks cycles, then immediately returns menu control to the calling *Display Options* Menu (M-4).

```
CONTROL OF INSTRUCTION QUEUEING AND BREAKPOINTS
MENU (M-4-22)

Any positive number N sets a break point at N clocks and returns to the
previous menu. Option "-1" turns off modeling of the VSP hardware
queue (default is off). The next three options turn on the hardware and
software queue models. In addition, they select when control is returned to
the host after an instruction queued. The instruction may or may not
complete execution. Explicit WAITs and queue full can force instruction
completion. Turning on the hardware queue also enables the software
queues unless they are specifically turned off with option "-7".

-1 Do not model queueing
-2 Return immediately to host (this defers instruction execution
   until an explicit WAIT or the queue is full)
-3 Complete each instruction immediately
-4 Specify random model parameters (each instruction will execute
   for a random number of clocks before returning)
-5 Reset VSP instruction execution and VSP queues

VSPS software queues including fetch queue:
-6 On          -7 Off

-8 Return to previous menu

Specify value of BREAK point or queueing option (-8)(-8):
```

Figure 4-9. Control of Instruction Queueing and Breakpoints Menu - Menu M-4-22.

4.4.5 VSP Simulator Signal Processing Library Menu - Menu M-5

The VSP simulator comes equipped with a number of common signal processing algorithms already programmed. Menu M-5 is shown in Figure 4-10 and shows which algorithms are already programmed into the simulator and what the terminal screen looks like for selecting these algorithms. The algorithm parameters can be "customized" by simulator users by interactively specifying queried parameters such as the length of data vectors and where these vectors exist in simulated external VSP memory. These algorithms are useful from the standpoint that they are already programmed and debugged and may be used as example programs for how to write VSP code. In addition, they are useful for evaluating VSPS accuracy by comparing their results with the same algorithms programmed in the IEEE Signal Processing Library.

The names of the algorithms shown in Menu M-5 are descriptive of what signal processing function is performed. Additional descriptions of the algorithms are provided below. The HELP option in the menu also goes into additional discussions of the algorithms.

```
Available processes:

1 Fast convolution/correlation
2 Complex vector add
3 Complex vector multiply
4 Power spectrum of complex data
6 Magnitude of complex vector
7 Moment of power spectrum of complex vector
8 General FFT (Fast Fourier Transform)
9 256-point demo FFT
10 Real FFT
11 HELP
0 Exit to Main Menu

Specify value of process (-1)(0):
```

Figure 4-10. VSP Simulator Signal Processing Libray Menu - Menu M-5.

Description of menu choices for Menu M-5:

1 Fast convolution/correlation

This technique calculates the convolution or correlation result of two data sequences in the frequency domain. Rather than calculating a time-domain convolution result, both sequences are transformed into the frequency domain, where an element-by-element multiplication rather than a convolution operation is performed. The multiplication result is inverse-transformed back to the time domain which corresponds to the convolution or correlation result desired.

Note that the term *fast convolution* is not meant to imply that the transform-multiplication-inverse-transform operation is always faster than the time-domain convolution operation; it is completely data dependent. However, for large convolution results, it may be more efficient to perform the calculations in the frequency domain and then perform an inverse-transform to return to the time domain.

2 Complex vector add

This signal processing option performs a complex vector addition using two vectors of arbitrary length existing in external memory. The real and imaginary parts of one vector are added to the corresponding real and imaginary part of the second vector. The resulting vector is stored back into external memory.

3 Complex vector multiply

This signal processing option performs a complex vector multiplication using two complex vectors of arbitrary length existing in external memory. Each complex sample in one vector is multiplied with the corresponding complex sample in the other vector. The result is stored back into external memory.

4 Power spectrum of complex data

This signal processing option calculates the power spectrum of a complex vector existing in external memory. The vector length must be an even power of two. The procedure involves first calculating the FFT of the complex input vector, then calculating the square of the magnitude of the resulting transformed vector. The result is stored back into external memory.

6 Magnitude of complex vector

This signal processing option calculates the magnitude of each complex sample existing in an arbitrary-length complex vector in external memory. The calculation is performed by first taking the absolute value of each complex sample in the vector which moves all samples to the first quadrant of the unit circle. Then each complex sample is successively rotated closer and closer towards the 0^o axis. In the limit, when the complex sample lies on the x-axis, the imaginary component is zero and the magnitude of the sample is equal to the x-component. After three vector rotations, the maximum error in magnitude is 1.9%. After five vector rotations, the maximum error in magnitude is 0.12%.

7 Moment of power spectrum of complex vector

This signal processing option calculates the moment of a power spectrum. The power spectrum is simply an FFT of a complex vector followed by an MGSQ instruction as in option '4'. The moment operation first performs a MLTR instruction on the output of the power spectrum with an external ramp function, then scales the accumulated sum (from the MLTR) by the total power accumulated by the MGSQ calculation.

The external vector used in the ramp is a "real" linear function corresponding to the *index* of the spectrum; i.e., 0,1,2,...127 for a 128-point spectrum. Note that the indices are scaled from 0 to 32767 corresponding to the full scale range of the VSP. Additionally, because the output addresses of the FFT are in bit-reversed order, the ordering of the ramp vector in external memory is also in bit-reversed order. It is not necessary for the user to provide the ramp vector; it is calculated and stored in external memory by the library function. The accumulated value of the MLTR instruction is scaled with the SCL instruction by the power in all the frequencies accumulated during the MGSQ operation.

8 General FFT

The general FFT signal processing option uses the VSP instruction set for calculating a general-length FFT from 2 to 4096 points. The input vector is assumed to be complex, and the calculation uses block floating-point scaling. The real and imaginary results of the FFT are stored back into external memory.

9 256-point demo FFT

The 256-point demo FFT calculates 256-point complex FFT on a complex vector which is defined as beginning at memory base address 0 in external memory. The real and imaginary results of the FFT calculation are written back out to external memory. There are no user parameters queried for this demo FFT.

10 Real FFT

The real FFT signal processing option is a powerful library function which provides two different types of FFT calculations. A "real FFT" means that the input vector is assumed to be a real vector, not a complex vector. An FFT calculated on real input data generates a symmetric spectrum about the N/2 transform point. The output of each real FFT calculation is a complex vector of length N/2 complex samples corresponding to the first half (unfolded) portion of the spectrum.

The first real FFT option calculates two simultaneous real FFTs using one complex FFT instruction. The real vectors are assumed to exist in external memory with a maximum length of N=128 real samples each. One of the real vectors is loaded into the real portion of VSP memory, and the other real vector is loaded into the imaginary portion of VSP memory. After the FFT calculation, two N/2 complex-point sequences corresponding to the transforms of the two input vectors are written to external memory.

The second real FFT option calculates a 2*N-point real FFT using an N-point complex FFT instruction. The input vector is assumed to exist in external VSP memory with a maximum length of N=256 real samples. After the FFT calculation, an N/2 complex-point sequence is written out to external memory corresponding to the first half of the symmetric spectrum.

4.4.6 IEEE Signal Processing Library Menu - Menu M-6

To facilitate algorithm comparisons between the VSP integer (or block floating-point) arithmetic and full floating-point arithmetic, a package of IEEE signal processing algorithms is included within the simulator. The algorithms included from the IEEE are shown in Figure 4-11 as Menu M-6. All of the programs in this menu are also included in the *VSP Simulator Signal Processing Library* (Menu M-5). The difference is that the IEEE routines are written using floating-point arithmetic, while the VSP signal processing library is written using the integer VSP instructions. The IEEE routines are particularly useful in making comparisons with VSPS programs for arithmetic accuracy. All queries in either menus M-5 and M-6 are the same for the corresponding library. There are no subordinate menus called from Menu M-6. The HELP selection provides the same text as the HELP selection for Menu M-5.

This program contains routines adapted, with permission, from the IEEE publication "Programs for Digital Signal Processing". ZORAN Corporation is solely responsible for the operation and support of this program.

```
Available processes:

1 Fast convolution/correlation
4 Power spectrum of complex data
6 Magnitude of complex vector
8 General FFT (Fast Fourier Transform)
10 Real FFT
11 HELP
0 Exit to Main Menu

Specify value of process (-1)(0):
```

Figure 4-11. IEEE Signal Processing Library Menu - Menu M-6.

4.4.7 Application Library Menu - Menu M-7

The applications library contains complete applications programs which have been included in the VSPS. Currently two applications exist in this library: Doppler-shift application and a 16K-point FFT. They are shown in Figure 4-12 and constitute Menu M-7. More applications will be included in the library with time. It is intended that the applications library exist as a resource to simulator users for reference and as a library from which appropriate VSP code may be extracted.

Discussions of the applications included in this menu occur in Chapter VIII.

```
APPLICATION LIBRARY
MAIN MENU
Select-

1 Doppler-shift application
2 16K FFT
10 Return to previous menu

Specify value of selection (0)(10):
From line 0 in "system": WAIT ALLDONE
Type 'RETURN' to continue:
```

Figure 4-12. Application Library Menu - Menu M-7.

Note: The application library is currently not provided in the PC version of the VSP simulator.

4.4.8 Break Menu

The simulator supports a *non-destructive* HELP feature which allows temporary interruption of the current processing. At any point in time while using the simulator, even during user program or library program execution, typing a 'ctl-C' (control C) will interrupt the current execution and bring the *Break* Menu into view. The upper half of the *Break* Menu resembles Menu M-4, and the lower half provides some obvious help functions. The *Break* Menu is shown in Figure 4-13.

From this menu, jumps may be made to other menus for changing or setting options, plotting or displaying memory, changing the message level, or aborting the current processing, or the simulator may be exited entirely. It is a very useful feature for halting or modifying program execution.

```
BREAK MENU
Message level determines the information
displayed during instruction execution.

Message levels are:
0  print nothing          1  print VSP instructions
2  print VSP instructions and wait for
   RETURN or selection from Display Options Menu
3  print detailed description of each instruction

11 Return to Main Menu (abort current processing)
12 Display options, message level and queueing
13 Data Generation and Display
14 Exit Simulator
16 Return to current processing

Setting the message level exits from this menu

Specify value of your selection (2):
```

Figure 4-13. Break Menu.

Note that on the PC version of the simulator, option 11 is not supported. It is not possible to abort the current processing and jump directly to the *Main* Menu. All other options are supported on the PC.

4.5 A First Session with the VSPS

This section will show a complete example of how to enter the VSPS environment, perform some very simple operations with the VSPS, and return to the operating system. By following the examples, a path will be traced through the menus which will make some of the simpler choices available. After this example, the user should have gained enough confidence to allow tracing different paths through the simulator and its menus while making different and more sophisticated choices. After working with this exercise, novice users are encouraged to experiment with different menu choices, as well as read the HELP selections available in each menu. A few such sessions should insure confidence in using the simulator and in learning the powerful features provided in the VSPS environment.

4.5.1 Entering the VSPS Environment

With the VSPS installed on the host computer, type 'vsps' <cr> on VAX computers or 'vsps' <cr> on PCs at the prompt. After a short pause, the system displays will disappear and the screen will display the *Main* Menu of the VSPS as shown in Figure 4-14. From the *Main* Menu, if the user wishes to create and store waveforms in simulated external memory, he should select option '3'<cr>, as shown in the figure. This selection calls the signal generator within the simulator which allows many types of real or complex signals to be created and stored in simulated external memory. The user is encouraged to use the '1' or HELP selection and read the material that this choice provides.

4.5.2 Creating and Storing a Signal In Simulated External Memory

The '3' option from the *Main* Menu selects the *Data Generation and Display* Menu. From this menu, an '8' <cr> will load a signal to simulated external memory. This selection calls the *Signal Generatio*n Menu, which allows signal generation and storage in simulated external VSP memory. In addition, the user is allowed to save signals to disk files. When a '2' <cr> is entered, the *normalization after summation signal generator* is called. The VSPS will pause for a few seconds before responding while it is initializing memory buffers. Presently the message shown as the first line in Figure 4-15 will appear:

"Specify value of memory base address:"

This is the beginning of the menu choices for creating a simulated signal; the successive prompts are also shown in Figure 4-15 as they appear on the terminal.

In this example terminal session, prompts will be responded to in most cases by selecting the default values; the result will be the loading of a *complex sine wave* starting at simulated external memory location *0* and continuing for a total of *128 complex samples*. Some of the prompts display two values before pausing for a return. To select the default value, (the second number in parentheses when two are present), simply type a <cr>. To go back in the prompt sequence and have the opportunity of re-entering a previous entry, type the first value displayed and then hit a <cr>. For example, the first prompt queries:

"Specify value of memory base address (-1) (0):"

Hitting a <cr> will specify that the signal about to be created will be loaded starting at physical memory address 0 in simulated external memory. Any other legal physical address in the 64K word address space could be provided as a response to the prompt also. Typing '1000' <cr> would begin the storage of the waveform at physical memory address 1000 decimal. If '-1' <cr> is typed, the VSPS will exit the signal generator and reposition the program to the previous *(Signal Generation)* menu. In this example, the default value of 0 is selected by typing a <cr>. We are now asked to:

"Specify value of maximum total amplitude (negative for real data) (1.00000):"

The default for this prompts is 1.0. If 1.0 is selected, the signal generated will be scaled to have a peak magnitude of 16384 in simulated external memory. In other words, external memory is scaled to its maximum integer range. Valid responses to this prompt are from -1.0 to +1.0. The magnitude of the number represents the amplitude of the signal created relative to the full-scale 16-bit integer range of the VSP. Negative values are a command to the signal generator to create a purely real signal with no imaginary component, and to write this real signal into consecutive 16-bit external memory locations. Positive numbers command the signal generator to create a complex signal with real and imaginary components, and to write the complex signal into external memory as complex pairs of real and imaginary samples. The absolute value of a negative entry also specifies the peak magnitude of the signal generated.

The next prompt queries:

"Specify value of number of samples (0)(128):"

The response to this prompt is used to specify the number of samples which should be created and written to simulated external memory. Depending on whether a real or complex signal is being generated, either N or 2*N entries (respectively) will be created and written to simulated external memory. A <cr> is typed to specify 128 complex samples (128 real and 128 imaginary words).

The next prompt:

"Specify value of 1 for bit-reversed, 2 clearing memory to 0 data (-1)(0):"

allows the storing of data in simulated external memory using bit-reversed addresses if '1' is selected. See the section on the FFT and bit-reversing for an explanation of bit-reversal in greater detail. Option '2' clears the number of memory samples specified, beginning at the memory base address already selected. Option '0' (the default selected in this example) generates data in normal addressing order by typing a <cr>.

At the next prompt:

> *"Specify value of 1 - Sum of Signals, 2 - Special, 3 - Saved. (0)(1):"*

the default value of 1 is selected by typing a <cr> to specify a signal that will be made from the summation of a series of signals that we will choose from the next prompt.

The next prompt:

> *"Signal options are:*
> *1 - Flat or Step, 2 - Impulse, 3 - Cosine*
> *4 - Uniform random, 5 - Square wave, 0 - to Quit*
> *Specify value of signal type (-1)(3):"*

allows the user to choose the type of signal to add to memory. In this example, the default of a cosine wave is adequate, and is selected by typing a <cr>.

The next prompt:

> *"Specify value of relative amplitude (1.00000):"*

allows the user to specify the relative amplitude of the signal generated in this pass to others created in additional passes which will be summed together to create the final compound waveform. In this example, only one signal will be generated, so the default value of 1.0 is selected by typing a <cr>.

The next prompt:

> *"Specify value of phase in degrees (30.00000):"*

allows the user to select the initial phase for the current waveform being generated. In this example, the default value of 30^{o} is acceptable; it is selected by typing a <cr>.

The next prompt:

> *"Specify value of cycles per total samples (1.00000):"*

allows specification of how many complete cycles of the specified waveform should be generated in the 128 samples we specified earlier. Because we would like to take an FFT on the signal we generate, choose an option other than the default value of 1.00000, such as '4' <cr>.

At this point, the screen displays the list of signal types once again. If another waveform is selected at the prompt, this signal will be added to the cosine wave already created and the user will be asked to select the phase, relative amplitude and number of cycles of this signal as well. This process can be continued until a sophisticated signal with many components had been created. In this example, option '0' is selected to exit the signal generation process; a '0' is selected to exit the *Signal Generation* Menu without saving the signal just created to a special memory buffer or to a disk file. The signal has, however, been stored in simulated external (complex) memory samples 0 through 127, or physical memory addresses 0 through 255. The screen should now display the *Signal Generatio*n Menu.

It is now necessary to return to the *Main* Menu because the next step in this example is to execute VSP instructions on the data just created and stored in simulated external memory. Selecting the default options from the *Signal Generation* Menu and the *Data Generation and Display* Menu will return program operation to the *Main* Menu.

4.5.3 Interactive Instruction Execution

Instructions can be executed one at a time in a tutorial environment by selecting option '2' from the *Main* Menu. Typing '2' <cr> selects the *Instruction Selection Tutorial* Menu as is shown in Figure 4-16. The bottom half of this menu lists the VSP instruction set. Selecting an instruction for execution is performed by simply typing the name of the instruction plus a carriage return at the prompt. In this particular example, the first step is to execute a load (LD) instruction on the data we have just generated which moves data in simulated external memory into internal VSP RAM. The first step in loading the data is to type 'LD' <cr> at the prompt as shown.

After 'LD' is typed, the menu shown in Figure 4-17 appears. The parameter values displayed in the upper half of the menu are the default parameters defined by the VSP simulator. These are the parameters which will be used when the LD instruction is executed unless changes are made to them. For this example, we want to load all 128 complex points that we have generated so we type 'nmpt' <cr> as shown; at the next prompt we type '128' <cr>. We have now set the NMPT field of the LD instruction to 128. All the other values are correct for this exercise, especially MBA:0 (memory base address:0) which points to address 0 of simulated external memory. A <cr> causes a reeturn to the previous menu, while a '0' <cr> causes the LD instruction and parameters to be displayed as shown in Figure 4-17; typing a second '0' <cr> causes the instruction to be executed.

Now that the complex sine wave has been loaded into the simulated VSP RAM, we want to examine this RAM to see what the data that it contains look like. To do this, we back up to the *Main* Menu and there we type '3' <cr> to select the *Data Generation and Display* Menu. Once we are in the display menu we choose option '12' to display VSP RAM and option '13' to display register contents. The screen will begin to scroll showing a display like that of Figure 4-19.

This is a display of the complex sine wave now contained in (complex) memory locations 0 through 127; each sample is indexed as VSPRAM:0 [number](real part, imaginary part). The "0" after VSPRAM: refers to the first simulated VSP in the system (recall that the VSPS can simulate systems where several VSPs share the same bus). The indices run from 0 to 127, indexing all of the complex samples which we generated using the signal generator.

Internal VSP RAM is 17 bits long; notice that the numbers in external memory appear to have been multiplied by two when they were transferred to internal RAM. They have in fact been bit-extended (left-shifted by one bit) to 17 bits. Data conversion from 16 to 17 bits and back again is handled automatically as data is moved from internal memory to RAM and back.

After the VSP RAM and register contents have been displayed, typing <cr> returns operation to the *Data Generation and Display* Menu. From there we return to the *Main* Menu, and from there we select option '2' to return to the *Instruction Selection Tutorial* Menu.

Now we are going to perform an FFT on the contents of internal VSP memory and display the result. Type 'fft' <cr> in the *Instruction Selection Tutorial* Menu to choose the FFT instruction. A full description of the 'FFT' instruction exists in section 6.7. For the purposes of this illustration we need to change the following parameters: NMBT (for number of butterflies) to logical 128 and FPS (first point separation) to 64. Now execute the FFT instruction just as the LD was executed by typing '0' plus two carriage returns; the instruction will execute and store the results in internal VSP RAM.

In order to make sense of the display of the results from the FFT instruction we need to bit-reverse the addresses of the output data so that they appear in normal order. To do this, we again use the instruction tutorial. Select the ST instruction, set MBA to 0 and RV to 1. Executing ST ('0' plus a double <cr>) now puts the bit-reversed ouput of the FFT back into simulated external VSP memory beginning at physical address 0. Since the FFT bit-reversed its results, the memory contents are now back in normal order after the ST instruction.

4.5.4 Plotting External Memory on the Terminal

The next order of business is to display the results. Return to the *Main* Menu and select option '3' for the *Data Generation and Display* Menu. From this menu select option '5' for a graphics display of memory contents. From this point, simply accept the default values which are prompted on the screen for: data type (which will be complex), a "single series" plot, base address (which will be zero), and the number of samples (which will be 128). After a few seconds the terminal will display the output which is just the spike of the single frequency value of the sine wave that was created in the beginning; this is shown in Figure 4-20.

Now type <cr> to erase the display on the screen, follow the prompt instructions to return to the *Data Generation and Display* Menu, then to the *Main* Menu and finally select option '11', which returns control to the operating system.

We have now entered the VSPS from the system, created a signal, loaded it from simulated external memory to simulated VSP RAM and displayed the RAM contents on the screen. From this point we did an FFT, stored the results in nomal order, displayed them, and backed out to the system.

You are now in a position to go in and repeat this process, this time progressively increasing the complexity of what you do by generating more complicated signals, operating on them in more complicated ways and choosing other display options. The VSPS is more than just a simulation and a teaching tool, it is a complete algorithm and system development environment. The more time spent with the simulator, the more will be learned about both the VSP and the ways in which the target application can be simulated.

Figure 4-14. VSPS Main Menu.

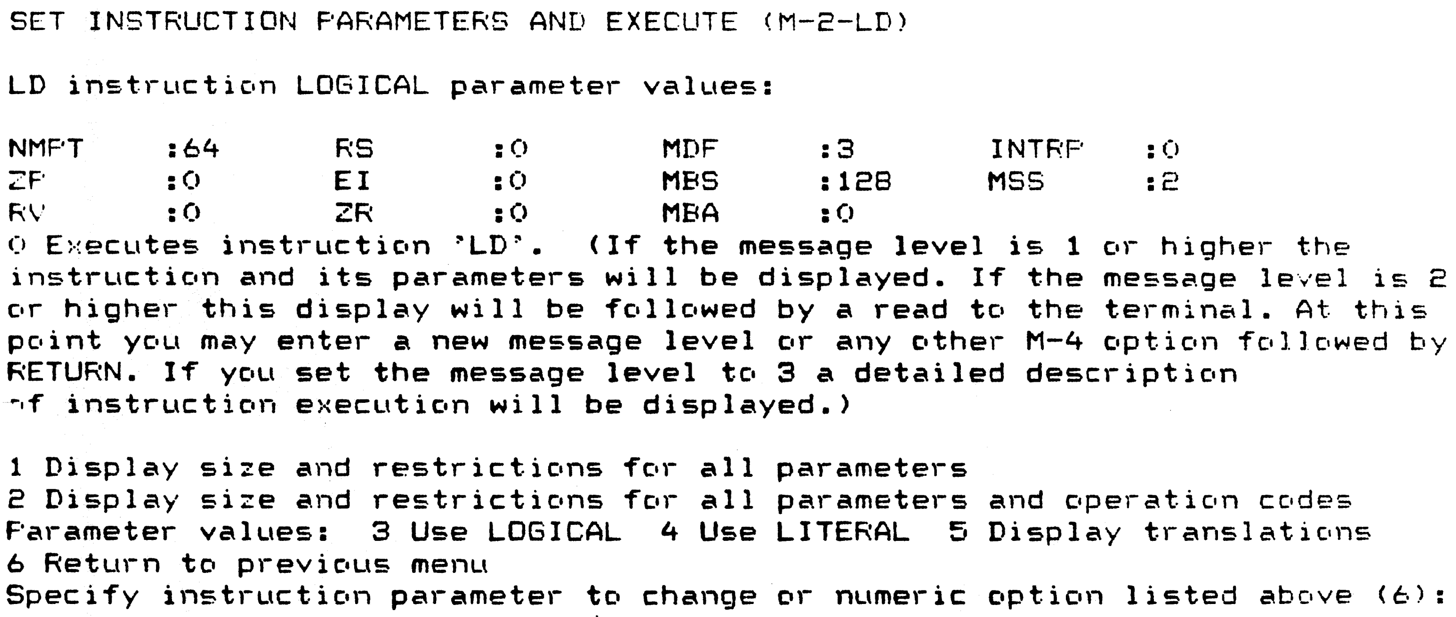
```
VECTOR SIGNAL PROCESSOR SIMULATOR FOR IBM PC/AT V2.35


ZORAN CORPORATION PROPRIETARY SOFTWARE
COPYRIGHT (C) 1986 ZORAN CORPORATION
ALL RIGHTS RESERVED

MAIN MENU

 1 HELP
 2 VSP instruction tutorial and execution
 3 Data generation and display
 4 Display options, timing control and queueing
 5 Signal processing library for VSP
 6 IEEE signal processing library
 7 Application library
 8 Execute user program in vspop()
 9 Execute batch commands and VSPS validation
10 Command mode
11 EXIT

Specify value of your selection (0)(1):
```

```
Specify value of memory base address (-1)(0):
Specify value of maximum total amplitude (negative for real data) ( 1.00000):
Specify value of number of samples (0)(128):
Specify value of 1 for bit reversed, 2 clearing memory to 0 (-1)(0):
Specify value of 1 - sum of signals, 2 - special, 3 - saved, (0)(1):
Signal options are:
 1- Flat or Step, 2- Impulse, 3- Cosine
 4 - uniform random, 5 - square wave, 0 - to quit
Specify value of signal type (-1)(3):
Specify value of relative amplitude ( 1.00000):
Specify value of phase in degrees (30.00000):
Specify value of cycles per total samples ( 1.00000):
Signal options are:
 1- Flat or Step, 2- Impulse, 3- Cosine
 4 - uniform random, 5 - square wave, 0 - to quit
Specify value of signal type (-1)(3):
Specify value of relative amplitude ( 1.00000):
Specify value of phase in degrees (30.00000):
Specify value of cycles per total samples ( 1.00000):
Signal options are:
 1- Flat or Step, 2- Impulse, 3- Cosine
 4 - uniform random, 5 - square wave, 0 - to quit
Specify value of signal type (-1)(3):
```

Figure 4-15. Prompts from Signal Generator.

```
INSTRUCTION TUTORIAL: SELECTION MENU (M-2)

Enter an instruction name or a numeric option

 1 HELP
 2 Set CYCMEM in mode register
 3 Set number of RAM sections in mode register
 4 Initialize scale nibbles
 5 Dump instruction parameter descriptions to file
 6 Set the VSP number
Set bit numbering:  7 right to left    8 left to right
VSP interrupts:     9 enable    10 disable
11 Modify External RAM
12 Return to previous menu

Instruction names are:
NOP        JMPI       LDSM       LD         STB
ST         STI        MLTC       MLTR       ADDC
ADDR       FFT        DEMO       MODLT      SCLT
SCL        ABS        CMCN       MGSQ       CMLT
ACCI       ACCR       HLT

Specify instruction or numeric option for instruction demonstration (12):
```

Figure 4-16. Instruction Selection Tutorial Menu (Menu M-2).



```
SET INSTRUCTION PARAMETERS AND EXECUTE (M-2-LD)

LD instruction LOGICAL parameter values:

NMPT     :64       RS       :0        MDF      :3        INTRP    :0
ZP       :0        EI       :0        MBS      :128      MSS      :2
RV       :0        ZR       :0        MBA      :0
0 Executes instruction 'LD'.  (If the message level is 1 or higher the
instruction and its parameters will be displayed. If the message level is 2
or higher this display will be followed by a read to the terminal. At this
point you may enter a new message level or any other M-4 option followed by
RETURN. If you set the message level to 3 a detailed description
of instruction execution will be displayed.)

1 Display size and restrictions for all parameters
2 Display size and restrictions for all parameters and operation codes
Parameter values:  3 Use LOGICAL  4 Use LITERAL  5 Display translations
6 Return to previous menu
Specify instruction parameter to change or numeric option listed above (6):
```

Figure 4-17. Instruction Execution Menu (Menu M-2-InOp).

```
LD    NMFT:128,RS:0,MDF:3,INTRP:0,ZP:0,EI:0,MBS:128,MSS:2,RV:0,ZR:0,MBA:0,
```

Figure 4-18. LD instruction and parameters prior to execution.

```
VSPRAM:0[  0  ]( 56762, 32772)
VSPRAM:0[  1  ]( 55086, 35518)
VSPRAM:0[  2  ]( 53276, 38178)
VSPRAM:0[  3  ]( 51338, 40746)
VSPRAM:0[  4  ]( 49278, 43216)
VSPRAM:0[  5  ]( 47098, 45582)
VSPRAM:0[  6  ]( 44804, 47838)
VSPRAM:0[  7  ]( 42404, 49978)
VSPRAM:0[  8  ]( 39900, 51998)
VSPRAM:0[  9  ]( 37300, 53894)
VSPRAM:0[ 10  ]( 34612, 55660)
VSPRAM:0[ 11  ]( 31838, 57290)
VSPRAM:0[ 12  ]( 28988, 58784)
VSPRAM:0[ 13  ]( 26070, 60136)
VSPRAM:0[ 14  ]( 23098, 61342)
VSPRAM:0[ 15  ]( 20050, 62400)
VSPRAM:0[ 16  ]( 16964, 63310)
VSPRAM:0[ 17  ]( 13836, 64066)
VSPRAM:0[ 18  ]( 10676, 64668)
VSPRAM:0[ 19  ](  7490, 65114)
VSPRAM:0[ 20  ](  4286, 65402)
VSPRAM:0[ 21  ](  1072, 65534)
VSPRAM:0[ 22  ]( -2144, 65508)
VSPRAM:0[ 23  ]( -5
```

Figure 4-19. Terminal dump of internal VSP RAM.

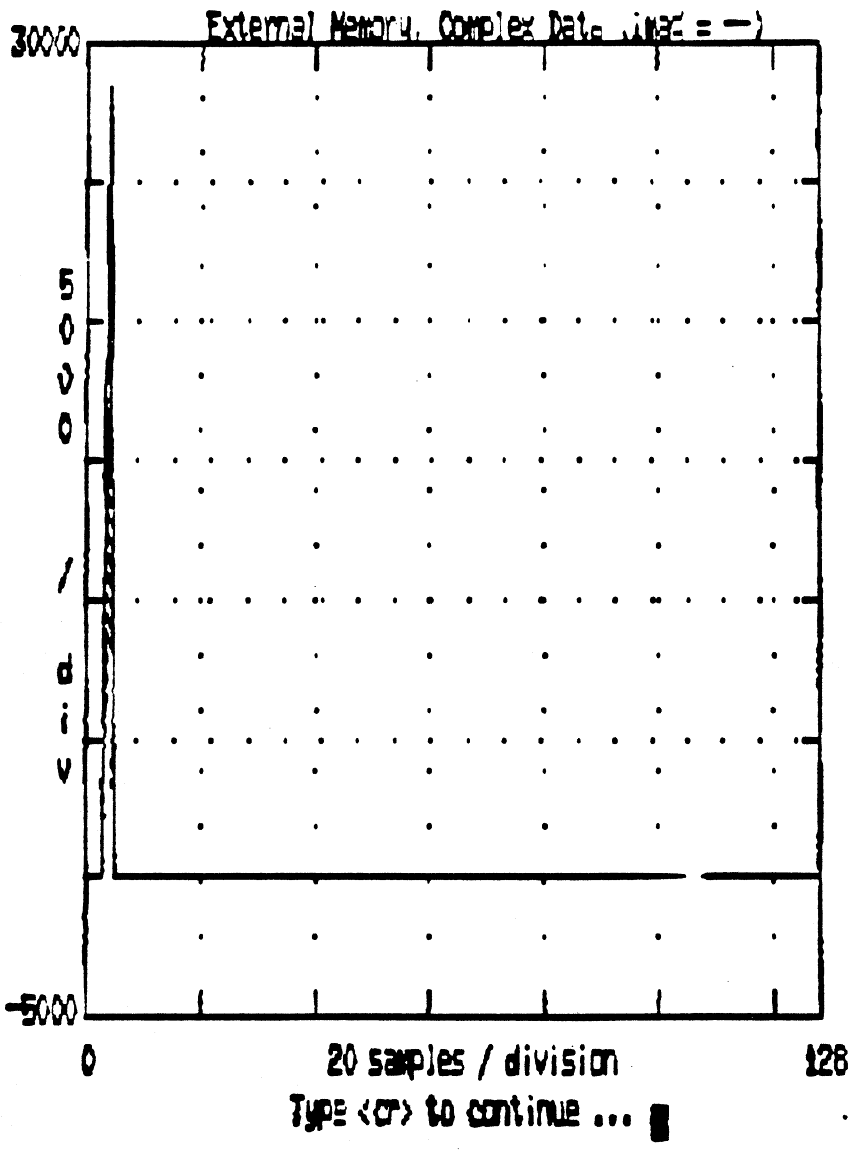

External Memory, Complex Data (imag = —)
30000
5000 / div
-5000
0
20 samples / division
128
Type <cr> to continue ...

Figure 4-20. FFT of Signal Generated in Example Terminal Session.

CHAPTER V

VSP REGISTERS

5.1 Overview

As mentioned earlier in Chapter II, the VSP system processor contains a number of registers in addition to the RAM and ROM memory areas. In order to use the VSP simulator properly, the user must become familiar with the function and usage of the registers. The mode register sets the operating mode of the VSP. There are eight 16-bit *information registers*, all of which can be read: *status, next fetch address, scale, maximum scale and two accumulators.* (The two accumulators are stored in four registers as each accumulator is longer than 16 bits.) Additionally, there are three *special-purpose registers*: the old maximum scale register, the instruction FIFO and the instruction base/start register. The information registers provide the status of the VSP, the scale factors in the FFT instruction, instructions yet to be executed, and results of vector operations in the accumulators.

Table 5-1 lists the unique addresses of the VSP registers. Most of the registers are either read-only or write-only. The exception to this is the scale register which can be both written to and read. Notice that the real and imaginary accumulators are accessible as two reads to separate VSP addresses because the accumulators are longer than 16 bits. The sine-cosine look-up table and RAM address spaces are also listed in this table to show how these addresses integrate with those of the registers.

TABLE 5-1 Tabular listing of the internal VSP registers and their addresses.

Register Name	Binary Address	Access
Mode	1100000000	(write)
FIFO (without execution)	1100000010	(write)
FIFO (execute immediately)	1100000011	(write)
Old Maximum Scale	1100000100	(write)
Instruction Base/Start	1100000110	(write)
Scale Vector	110001XXXX	(read/write)
Next Fetch Address	1100000000	(read)
Status	1100000001	(read)
Maximum Scale	1100000010	(read)
Scale	1100000011	(read)
Imaginary Accum (MSB)	1100000100	(read)
Imaginary Accum (LSB)	1100000101	(read)
Real Accum (MSB)	1100000110	(read)
Real Accum (LSB)	1100000111	(read)
RAM	00XXXXXXXX	(read/write)
[1]Sine-cosine tables	01XXXXXXXX	(read)

[1]Note that the Sine-cosine tables are not externally addressable (readable). They are used for internal calculations only.

The format and description of the VSP registers are provided in the following sections.

5.2 Mode Register

This register programs the operating mode of the VSP. It is 16 bits in length, and contains the following parameter format:

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
|RST|RSS|SYN|NCS|NMS| 0 |CRQ| 0 |INL|   CYCMEM    |IDO| 0 |ILI|IFO|
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The names and functions of each of the mode register parameters are given below:

RST (Reset) RST resets the VSP when its value is 1. The function is the same as that of the external RESET\ pin.

RSS (Restart Scale) RSS resets the maximum scale register and the pointer to the scale register. This causes the next scale factor to be written to the least significant nibble and maximum scale accumulation to restart.

SYN (Synchronization) SYN determines whether input control signals RD\, WR\, BACK\ and SUS operate synchronously or asynchronously: 1 for synchronous, 0 for asynchronous. After a reset, this bit assumes the synchronous state.

NCS (Number of Clocks per Cycle) NCS programs the number of internal VSP clocks required per cycle for external memory access when the system is in the VSP addressing mode: 1 for one clock per cycle, 0 for two clocks per cycle. This parameter is not used for instruction fetch; instruction fetch always assumes two clocks per memory fetch. NCS is programmed to 0 after a reset.

NMS (Number of RAM sections) NMS programs the number of sections for internal VSP RAM: 1 for one section, 0 for two. Two RAM sections are used for concurrent operation where the EU executes ALU instructions in one section while the BIU simultaneously executes I/O instructions in the other section. NMS is programmed to 1 after a reset.

CRQ (Continuous Bus Request) Determines whether the BRQ\ pin will become inactive for one ICLK between consecutive memory instructions (LD, LDSM, ST, STI, and
STB) or not. CRQ=1 will maintain BRQ\ enabled between consecutive memory instructions. CRQ=0 ensures that BRQ\ will be inactive for one ICLK between consecutive memory instructions. CRQ is programmed to 1 after a reset.

INL (Instruction Length) INL controls whether instructions are used with their customary length, which varies from one to three words, or with a fixed length of three words. HLT, the one exception, is always a two-word instruction. INL=0 programs variable length instructions, while INL=1 programs fixed three-word instruction lengths. INL is programmed to 0 after a reset.

CYCMEM (Cyclic Memory) CYCMEM defines the memory block size used for instructions which access external memory. When the VSP reaches the end of this memory block during a load or store, it loops back to the beginning of the block. The value of the three bits in binary is added to 9 to give the power of two representing the number of words in the memory block.

000	->	2**(9+0)	->	512-word blocks
111	->	2**(9+7)	->	64K-word blocks

For example, the following instruction:

LD NMPT=8, MBA=1020

executed while CYCMEM:0 in the mode register defines a cyclic memory block size of 512 words. When the instruction is executed, eight samples will be loaded from memory with the following addresses: 1020, 1021, 1022, 1023, 512, 513, 514 and 515.

CYCMEM is programmed to 111 binary after a reset.

IDO (Interrupt on Data Overflow) IDO=1 enables an interrupt when an overflow occurs in the ALU. IDO is programmed to 0 after a reset.

ILI (Interrupt on Last Instruction) ILI=1 enables an interrupt after the last instruction in the FIFO is executed, the instruction queue is empty and the device goes idle. ILI is programmed to 0 after a reset.

IFO (Interrupt on FIFO Overflow) IF0=1 enables an interrupt when a new instruction is written to the VSP when the FIFO is full. The content of the FIFO is not affected, and the VSP ignores the new instruction. IFO is programmed to 0 after a reset.

5.3 Status Register

This 16-bit register reflects all status information about the VSP except for scaling. It has the following format:

```
+---.---.---.---|---.---.---.---!---.---.---.---!---.---.---.---+
|       ARINSCO       |MIC| FIFOSTAT  |     |IMI|IAI|IDO| 0 |ILI|IFO|
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

ARINSCO (Arithmetic Instruction Code). ARINSCO represents five bits which hold the operation code of the last arithmetic instruction executed. The operation code is the first five bits of the first word of each instruction.

MIC (Move Instruction Code). MIC is coded to show the last move instruction executed as shown below:

0	->	LD, LDSM or JMPI
1	->	ST, STI or STB

FIFOSTAT (FIFO Status). FIFOSTAT provides a binary number from zero to four indicating the number of available instruction slots in the instruction FIFO.

IMI (Interrupt on Move Instruction). IMI is set at the completion of a move instruction.

IAI (Interrupt on Arithmetic Instruction). IAI is set at the completion of an arithmetic instruction.

IDO, ILI and IFO in the Status Register are flags controlled by the enabling parameters of the same name in the Mode Register. Along with the two other interrupt flags directly above (IMI and IAI), they are reset when the status register is read. The other bits are updated as befits their functions, as described in the preceding section on the mode register.

5.4 Instruction FIFO

The instruction FIFO is a 12-word buffer for holding instructions to be executed. It can hold up to four instructions, each consisting of as many as three 16-bit words. FIFO status is reported by the status register. Instructions fetched by the VSP are written into the FIFO by the BIU. Instruction fetch is initiated either by writing a JMPI instruction to the fetch queue or writing an address to the Instruction Base/Start register.

The host controller also has the ability to write instructions directly to the instruction FIFO independently of the BIU. If the host writes the instruction to VSP address 302H, it will be loaded into the VSP FIFO and will not be executed until VSP execution is enabled. This can be used for manually filling the VSP instruction FIFO prior to execution. Up to three instructions may be buffered by the host in the VSP FIFO in this manner. If instructions are written to VSP address 303H, the VSP will begin execution of the instructions contained in the FIFO. If no instructions previously existed in the FIFO, the instruction just written to the FIFO will be executed.

The instruction length (INL) bit in the mode register defines the length of the instructions to the BIU and instruction FIFO. Instruction word lengths may vary from one to three words in length. *Regardless of the individual length of instructions stored in the FIFO, the FIFO will hold a maximum of four instructions.*

5.5 Instruction Base/Start Register

This 16-bit register holds the base address of the program code to be executed in external memory. Writing an address to this register commands the VSP to begin instruction fetch and execution at the base address specified in the register.

5.6 Next Fetch Address Register

This 16-bit register holds the address of the next instruction to be fetched. It may be read by the host by reading its memory mapped address or by using the STI instruction.

5.7 Scale Register

The scale register is 16 bits in length and is partitioned into four four-bit nibbles. It holds the last four scale factors created by the execution of the FFT instruction. The scale factor is the number of right-shifts, (or divide-by-twos) performed during the execution of the FFT to prevent overflow. The scale factor is automatically stored in the register after FFT execution. A scale register pointer indicates which nibble will store the next scale factor. After each new scale factor is stored, the pointer is incremented to indicate the next nibble. The pointer reverts to the least significant nibble after a reset or after the most significant nibble is used. The register and the pointer are both reset by the Restart Scale flag (RSS) in the mode register and by the Load Scale parameter in the LDSM instruction.

5.8 Maximum Scale Register

The maximum scale register is a four-bit register containing the largest scale factor written since the last scale reset. It is read as a 16-bit register with all four nibbles equal to the maximum scale register. It may be reset by the *reset scale bit* in the mode register and by the *load scale* parameter in the LDSM instruction.

5.9 Old Maximum Scale Register

The old maximum scale register is a four-bit register containing the *previous* maximum scale factor, prior to the most recent reset. It can be loaded from the maximum scale register using the LDSM command. The value in the register is used with the SCL instruction to scale data vectors stored in internal VSP memory. (Use of this register will be explained in greater detail in later sections of the manual.)

5.10 Accumulators

The VSP has separate real and imaginary accumulators, each 25 bits in length. They are used for storing the results of vector arithmetic operations. The accumulators are accessible to the external world as the 24 most significant bits of each 25-bit word. Each accumulator is read as two 16-bit words. The least significant word holds the 16 least significant bits of arithmetic data. In the most significant word, the least significant byte consists of the upper eight bits of the accumulator; the most significant byte consists of eight sign-extended or virtual bits. The most significant word holds any overflow bits created during the accumulation process.

The real accumulator is updated by the following instructions: MLTR, MLTC, DEMO, MGSQ, CMLT, ACCR and SCL.

The imaginary accumulator is updated by MLTR, MLTC, DEMO, ACCI and SCL.

When an instruction uses an accumulator, it first clears the accumulator; instructions not requiring use of an accumulator will not affect that accumulator.

CHAPTER VI

VSP INSTRUCTION SET

6.1 Overview

This chapter covers the complete VSP instruction set. It is divided into five major sections, which cover specific types of instructions: memory, two types of ALU control, and the FFT. Each of the VSP instructions is covered in detail in one of the five sections. Note that the fifth section covers only the FFT instruction. This is because more detail and time is required to understand this powerful instruction and its parameters.

Each instruction and its respective parameter format is covered individually in one of the sections. In the format blocks for each instruction, fixed bits are shown as 0 or 1; the variable blocks, or parameters, are shown by label (field name); and the DON'T CARE bits are left blank. Multiple-word instructions are numbered from 0 to 2 when all three words are used. The bits in each word are numbered from 0 on the right end to 15 on the left end; the LSB is on the right and the MSB is on the left.

Parameters which are common to a majority of instructions are listed in section 6.2. This eliminates the duplication of discussing these parameters in each instruction.

Examples accompany all instructions and illustrate how the instructions affect both internal and external VSP memory, as well as VSP registers. Indexing conventions used in the VSPS are also given in these examples.

6.2 Common Instruction Parameters

This section lists instruction parameters which are common to a large number of instructions. The parameters are listed alphabetically for quick reference. Reference to this section is made throughout Chapter VI when these parameters are included in an instruction field.

AD - Addressing

0 -> VSP addressing mode
1 -> Host address generation mode

AD=0 is the normal VSP addressing mode. The VSP generates addresses and fetches its own instructions and/or data on the separate address and data buses.

When AD is programmed to 1, the VSP address bus assumes a high-impedance state. It is assumed that an external device generates the appropriate addresses to external memory. The data bus operates independently of the AD bit for both loading and storing data.

ADF - Arithmetic Data Format: determines which part of the operation result is stored in internal RAM

00 -> no change; the result goes only to the accumulators
01 -> imaginary part only stored
10 -> real part only stored
11 -> complete result is stored

CN - Constant: used to specify a constant in external memory, instead of a vector, to operate with the complex number in internal RAM.

0 -> add a vector
1 -> add a constant

EI - Enable Interrupt

0 -> no interrupt - only the status bit will be set
1 -> interrupt generated at end of instruction execution

MBA - Memory Base Address: the starting address of the data in external memory. This parameter is constant, and must conform to the convention described in Section 4.3.4.

MBS - Memory Block Size: number of real, imaginary or complex data points to be loaded before a skip occurs. See also RV below.

Literal		Logical
000	->	1 point
111	->	128 points

Logical 2^N is translated to N literal.

MDF - Memory Data Format for VSP internal RAM.

00	->	not used
01	->	imaginary only
10	->	real only
11	->	complex; first part real, second part imaginary

MSS - Memory Step Size: number of points specified in MBS plus the number of points to be skipped.

Literal		Logical
000	->	2 points
111	->	256 points

Logical 2^(N+1) is translated to N literal.

NMPT - NMPT defines the number of points (samples) of real, imaginary or complex data.

Literal		Logical
000 0001	->	1 point
111 1111	->	127 points
000 0000	->	128 points

Logical 128 is translated to 0 literal.

RS - RAM Section number

0	->	section 0	-	VSPRAM addresses 0 to 63 when NMS = 0
0	->	section 0	-	VSPRAM addresses 0 to 127 when NMS = 1
1	->	section 1	-	VSPRAM addresses 64 to 127 when NMS = 0

This parameter is used with the NMS parameter in the mode register to execute arithmetic and I/O instructions concurrently. RS can be 1 only if NMS=0 (specifying two RAM sections). If an ALU instruction operates with RS=0, a memory instruction with RS=1 can operate in parallel with the ALU instruction.

RV - Reverse ordering of data after being loaded into VSP internal RAM.

00	->	data in normal order
01	->	bit-reverse order one level
10	->	data within blocks of size MBS in normal order, blocks in bit-reversed order
11	->	data within blocks of size MBS in bit- reversed order, blocks in normal order

If RV = 01 or 10, NMPT must be a power of two.
If RV = 10 or 11, MBS is used for both memory segmentation and reversing - use caution. If RV is used for reversing only, set MSS = MBS - 1.

Examples 1 - 3 in section 6.3.1 illustrate the power of the RV parameter. A description is provided in each example of how the RV parameter affects the instruction.

Figure 6.1 below illustrates how the VSP implements bit-reversal.

Normal Order		Bit-Reversed Order	
Decimal	Binary	Binary Mirror Image	Decimal
0	00000	00000	0
1	00001	10000	16
2	00010	01000	8
3	00011	11000	24
4	00100	00100	4
5	00101	10100	20
6	00110	01100	12
7	00111	11100	28

Figure 6-1. Bit-Reversal.

SH - Shift

0 -> result is not right-shifted

1 -> result is right-shifted one bit, or scaled down by two, to avoid overflow

6.3 Memory Instructions

Memory instructions move data between external VSP memory and internal VSP memory or registers. All memory instructions are three words in length. The instructions covered in this section are:

LD (Load)
LDSM (Load Scale/Mode Register)
ST (Store)
STI (Store Information)
STB (Store Backward)

6.3.1 LD (Load)

LD moves data existing in external memory to internal VSP RAM. LD is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 0 | 0 |             NMPT              |RS | INTRP |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|    MBS    |    MSS    | RV    |AD |  MDF  |ZR |ZP |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters have definitions which are the same as those given in section 6.2: NMPT, RS, EI, MBS, MSS, RV, AD, MDF and MBA.

Parameters which are unique to the LD command:

INTRP - Interpolation: the number of zeros to be added after each data point is read from external memory.

00 -> no zeros added
01 -> one zero added - NMPT must be even
10 -> two zeros added - NMPT must be divisible by three
11 -> three zeros added - NMPT must be divisible by four

When using the INTRP parameter, NMPT *includes* the zeros to be added by INTRP. If zero padding (ZP=1), the constraints on NMPT above *do not* apply. If the data is complex, a real zero and an imaginary zero are added for each zero added by INTRP.

ZR - Zero Fill: used to fill a specified block of VSP internal RAM with zeros. One of the two MDF bits must be zero to indicate which part of the data, real or imaginary, will be filled.

0 -> internal memory unaffected
1 with MDF = 01 -> real part filled with zeros
1 with MDF = 10 -> imaginary part filled with zeros

If MDF = 11, ZR must be zero

ZP - Zero Padding - allows reading in a vector from external memory, then padding zeros to the end of the vector. When ZP=1, NMPT must be equal to the length of the external vector plus the number of zeros to pad.

0 -> no zero padding to end of vector
1 -> NMPT/2 or (NMPT + 1)/2 points from memory, the rest zeros

NOTE: When LD is used in the host address generation mode (AD=1), the following parameters are not used; their values are DON'T CARE: INTRP, MSS, ZP and MBA.

Three examples follow on how to use the LD instruction with different parameters.

EXAMPLE 1:

LD NMPT:8 RS:0 MDF:3 INTRP:1 ZP:0 EI:0 MBS:128 MSS:128 RV:0 ZR:0 AD:0 MBA:64

Before the LD instruction	After the LD instruction
ExtRAM [32] = [D64, D65]	VSPRAM [0] = [D64, D65]
ExtRAM [33] = [D66, D67]	VSPRAM [1] = [0, 0]
ExtRAM [34] = [D68, D69]	VSPRAM [2] = [D66, D67]
ExtRAM [35] = [D70, D71]	VSPRAM [3] = [0, 0]
ExtRAM [36] = [D72, D73]	VSPRAM [4] = [D68, D69]
ExtRAM [37] = [D74, D75]	VSPRAM [5] = [0, 0]
ExtRAM [38] = [D76, D77]	VSPRAM [6] = [D70, D71]
ExtRAM [39] = [D78, D79]	VSPRAM [7] = [0, 0]

Address	Ext RAM
64	D64
65	D65
66	D66
67	D67
68	D68
69	D69
70	D70
71	D71
72	D72
73	D73
74	D74
75	D75
76	D76
77	D77
78	D78
79	D79
80	D80
81	D81

Address	VSP RAM	
0	D64	D65
1	0	0
2	D66	D67
3	0	0
4	D68	D69
5	0	0
6	D70	D71
7	0	0

Example 1 loads eight complex words from external memory into the VSP RAM section 0 beginning at physical address 64. Zeros are included between each complex word because INTRP=1. NMPT includes the number of zeros added by the interpolation parameter.

EXAMPLE 2:

LD NMPT:8 RS:1 MDF:2 INTRP:0 ZP:0 EI:0 MBS:8 MSS:8 RV:1 ZR:0 AD:0 MBA:64

Before the LD instruction			After the LD instruction		
ExtRAM [32]	=	[D64, D65]	VSPRAM [64]	=	[D64]
ExtRAM [32.1]	=	[D66, D67]	VSPRAM [65]	=	[D68]
ExtRAM [33]	=	[D68, D69]	VSPRAM [66]	=	[D66]
ExtRAM [33.1]	=	[D70, D71]	VSPRAM [67]	=	[D70]
ExtRAM [34]	=	[D72, D73]	VSPRAM [68]	=	[D65]
ExtRAM [34.1]	=	[D74, D75]	VSPRAM [69]	=	[D69]
ExtRAM [35]	=	[D76, D77]	VSPRAM [70]	=	[D67]
ExtRAM [35.1]	=	[D78, D79]	VSPRAM [71]	=	[D71]

Address	Ext RAM
64	D64
65	D65
66	D66
67	D67
68	D68
69	D69
70	D70
71	D71
72	D72
73	D73
74	D74
75	D75
76	D76
77	D77
78	D78
79	D79
80	D80
81	D81

Address	VSP RAM	
64	D64	
65	D68	
66	D66	
67	D70	
68	D65	
69	D69	
70	D67	
71	D71	

Example 2 loads eight consecutive values from external memory into the real portion of VSP RAM section 1. The imaginary portion of memory is not affected. The NMS bit in the mode register must be programmed to 0 to allow loading RAM section 1. The eight samples read from external memory are stored in internal memory in bit-reversed order. In other words,

External Address 100(000) -> internal address 100(000)
External Address 100(001) -> internal address 100(100)
. .
. .
. .
External Address 100(111) -> internal address 100(111)

EXAMPLE 3:

LD NMPT:8 RS:0 MDF:1 INTRP:0 ZP:0 EI:0 MBS:2 MSS:2 RV:2 ZR:1 AD:0 MBA:65

Before the LD instruction			After the LD instruction		
ExtRAM [32.1]	=	[D65]	VSPRAM [0.1]	=	[D65]
ExtRAM [33]	=	[D66]	VSPRAM [1.1]	=	[D66]
ExtRAM [33.1]	=	[D67]	VSPRAM [2.1]	=	[D69]
ExtRAM [34]	=	[D68]	VSPRAM [3.1]	=	[D70]
ExtRAM [34.1]	=	[D69]	VSPRAM [4.1]	=	[D67]
ExtRAM [35]	=	[D70]	VSPRAM [5.1]	=	[D68]
ExtRAM [35.1]	=	[D71]	VSPRAM [6.1]	=	[D71]
ExtRAM [36]	=	[D72]	VSPRAM [7.1]	=	[D72]

Address	**Ext RAM**
64	D64
65	D65
66	D66
67	D67
68	D68
69	D69
70	D70
71	D71
72	D72
73	D73
74	D74
75	D75
76	D76
77	D77
78	D78
79	D79
80	D80
81	D81

Address	**VSP RAM**	
0	0	D65
1	0	D66
2	0	D69
3	0	D70
4	0	D67
5	0	D68
6	0	D71
7	0	D72

The third example loads the imaginary portion of internal VSP RAM section 0 with eight consecutive values from external memory beginning at address 65. The real portion of memory is filled with zeros because of the combination of the MDF bit and the ZR bit in the instruction. Because of the MSS parameter, the eight words are partitioned into four blocks of two words each. In conjunction with the RV parameter, the addressing of the *four blocks* is stored in bit-reversed order, but the data within the blocks is stored in normal order.

6.3.2 LDSM (Load Scale/Mode Registers)

LDSM moves data from external memory to the VSP's 64-nibble scale RAM or to the mode register, as determined by the MD bit in the instruction. It resets the maximum scale register and pointer to the scale register. LDSM also updates the old maximum scale register if MD=0 and UP=1, so that the next scale factor will be written to the least significant nibble, and maximum scale accumulation will be restarted.

LDSM is used only in the VSP addressing mode. It is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 0 | 0 |           NMPT            |RS | 0 | 0 |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 0 | 1 |           |UP |MD | 0 | 1 | 1 | 1 | 0 |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters have definitions which are the same as those given in section 6.2: NMPT, RS, EI and MBA.

Parameters which are unique to the LDSM command:

UP (Update): active only if MD = 0
- 0 -> no update, scale register pointer not reset
- 1 -> old maximum scale register updated from the current scale register; scale register pointer reset

MD (Mode)
- 0 -> only the scale registers and scale RAM are loaded
- 1 -> only the mode register is loaded

The following is an example of how the LDSM command may by used. Note that scale RAM is indexed starting at 140.

LDSM NMPT:8 RS:0 EI:0 UP:1 MD:0 MBA:21

Before the LDSM instruction			After the LDSM instruction		
ExtRAM [10.1]	=	[D21]	Scale [140]	=	[D21]
ExtRAM [11]	=	[D22]	Scale [140.1]	=	[D22]
ExtRAM [11.1]	=	[D23]	Scale [141]	=	[D23]
ExtRAM [12]	=	[D24]	Scale [141.1]	=	[D24]
ExtRAM [12.1]	=	[D25]	Scale [142]	=	[D25]
ExtRAM [13]	=	[D26]	Scale [142.1]	=	[D26]
ExtRAM [13.1]	=	[D27]	Scale [143]	=	[D27]
ExtRAM [14]	=	[D28]	Scale [143.1]	=	[D28]

```
Scale Ram - 16 bits      ExtRAM               SclRAM     ExtRAM
  First 8 words        Addr 16 bits         16 bits  Addr 16  bits
     _____                 _____             _____        _____
    |_____|            21 |_D21_|           |_D21_| 21  |_D21_|
    |_____|            22 |_D22_|           |_D22_| 22  |_D22_|
    |_____|            23 |_D23_|           |_D23_| 23  |_D23_|
    |_____|            24 |_D24_|           |_D24_| 24  |_D24_|
    |_____|            25 |_D25_|           |_D25_| 25  |_D25_|
    |_____|            26 |_D26_|           |_D26_| 26  |_D26_|
    |_____|            27 |_D27_|           |_D27_| 27  |_D27_|
    |_____|            28 |_D28_|           |_D28_| 28  |_D28_|

    Max Scl Reg           Old MSR              Max Scl Reg       Old MSR
+----.----.----.----+   +-----+            +---.---.---.---+   +----+
| MS | MS | MS | MS |   | OMS |            | 0 | 0 | 0 | 0 |   | MS |
+----.----.----.----+   +-----+            +---.---.---.---+   +----+

Scl Reg Ptr      Scale Register          Scl Reg Ptr  Scale Register
  +-----+   +----.----.----.----+          +---+    +---.---.---.---+
  | Ptr |   | S4 | S3 | S2 | S1 |          | 0 |    | 0 | 0 | 0 | 0 |
  +-----+   +----.----.----.----+          +---+    +---.---.---.---+
```

Referring to the LDSM example above, values existing in the internal scale RAM are undefined. After instruction execution, the scale RAM is loaded with scale values existing in external memory beginning at physical address 21. External memory remains the same. The maximum scale register, scale register pointer and scale register all contain old values which are cleared upon instruction execution. The old maximum scale register is loaded with the previous value stored in the maximum scale register.

6.3.3 ST (Store)

ST moves data from the VSP internal RAM to external memory. ST is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 0 | 1 |             NMPT                |RS |       |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   MBS     |   MSS     | RV    |AD |  MDF  | 0 | 0 |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All parameters defined in the ST instruction have the same definitions and use as in the LD instruction. Their descriptions are contained in section 6.2.

The following is an example of how the ST command may by used.

ST NMPT:8 RS:1 MDF:3 EI:0 MBS:2 MSS:4 RV:0 AD:0 MBA:41

Before the ST instruction	After the ST instruction
VSPRAM [64] = [R0, I0]	ExtRAM [20.1] = [R0, I0]
VSPRAM [65] = [R1, I1]	ExtRAM [21.1] = [R1, I1]
VSPRAM [66] = [R2, I2]	ExtRAM [24.1] = [R2, I2]
VSPRAM [67] = [R3, I3]	ExtRAM [25.1] = [R3, I3]
VSPRAM [68] = [R4, I4]	ExtRAM [28.1] = [R4, I4]
VSPRAM [69] = [R5, I5]	ExtRAM [29.1] = [R5, I5]
VSPRAM [70] = [R6, I6]	ExtRAM [32.1] = [R6, I6]
VSPRAM [71] = [R7, I7]	ExtRAM [33.1] = [R7, I7]

```
              VSPRAM                              MBA     ExtRAM
       ___________________                               ________
64    |___R0___|___I0____|                        41    |___R0___|
65    |___R1___|___I1____|                        42    |___I0___|
66    |___R2___|___I2____|                        43    |___R1___|
67    |___R3___|___I3____|                        44    |___I1___|
68    |___R4___|___I4____|                        45    |________|
69    |___R5___|___I5____|                        46    |________|
70    |___R6___|___I6____|                        47    |________|
71    |___R7___|___I7____|                        48    |________|
                                                  49    |___R2___|
                                                  50    |___I2___|
                                                  51    |___R3___|
                                                  52    |___I3___|
                                                  53    |________|
                                                  54    |________|
                                                  55    |________|
                                                  56    |________|
                                                  57    |___R4___|
                                                  58    |___I4___|
                                                  59    |___R5___|
                                                  60    |___I5___|
                                                  61    |________|
                                                  62    |________|
                                                  63    |________|
                                                  64    |________|
                                                  65    |___R6___|
                                                  66    |___I6___|
                                                  67    |___R7___|
                                                  68    |___I7___|
```

The ST example moves eight complex samples from VSP RAM section 1 into external memory beginning at physical address 41. The MBS parameter defines external memory as having a block size of two complex words. The MSS parameter defines the memory step size as four complex words. Thus, after two complex words are written to external memory, two additional complex memory locations are skipped. Note that MSS = MBS + skip size.

6.3.4 STI (Store Information Registers)

STI moves the contents of specified information registers within the VSP to external memory; it is used only in the VSP addressing mode.

STI is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 0 | 1 |           NMPT            |RS |       |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|    STR    |               |OR | 0 | 1 | 0 | 1 | 0 |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                             MBA                               |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters have definitions which are the same as those given in section 6.2: RS, EI and MBA.

Parameters which are unique to the STI instruction:

NMPT - Number of *information registers* to be read. Note that this meaning is different from the one used in earlier instructions. Values can be one through eight decimal, expressed as 000 0001 through 000 1000 binary.

STR - Starting Register: the place in the following list where the count of NMPT registers starts.

The logical numbers and names of the VSP registers are shown below for use with the STR parameter:

1 - Real Accumulator, LSB
2 - Real Accumulator, MSB
3 - Imaginary Accumulator, LSB
4 - Imaginary Accumulator, MSB
5 - Scale Register
6 - Maximum Scale Register
7 - Status Register
8 - Next Fetch Address

Literal		1st register stored (logical)	max(NMPT)
011	->	begin storage with reg #1	8
010	->	begin storage with reg #5	4
001	->	begin storage with reg #7	2
000	->	begin storage with reg #8	1

Logical to literal translations: 1->3, 5->2, 7->1, 8->0 literal.

There is an implied relation between NMPT (which defines the number of registers to store) and STR (which defines the register with which to begin the storage). This relationship is shown in the table above. For instance, if storage begins with logical register number 5 (scale register), the maximum number of registers which can be stored is 4. This is because the register address counter will not roll over.

OR - Order: arrangement of the list of registers above

0 -> numbers 2 and 3 are interchanged

1 -> the order above stands unchanged

The following is an example of how the STI instruction may be used. Note that the indexing of the information registers begins at 132.1.

STI NMPT:8 RS:0 EI:0 STR:1 OR:1 MBA:32

Before the STI instruction			After the STI instruction		
ACCRLSB [136]	=	[I1]	ExtRAM[16]	=	[I1]
ACCRMSB [135.1]	=	[I2]	ExtRAM[16.1]	=	[I2]
ACCILSB [135]	=	[I3]	ExtRAM[17]	=	[I3]
ACCIMSB [134.1]	=	[I4]	ExtRAM[17.1]	=	[I4]
SCALE [134]	=	[I5]	ExtRAM[18]	=	[I5]
MAXSC [133.1]	=	[I6]	ExtRAM[18.1]	=	[I6]
STATUS [133]	=	[I7]	ExtRAM[19]	=	[I7]
FETCHADR [132.1]	=	[I8]	ExtRAM[19.1]	=	[I8]

	Info Register	MBA	ExtRAM		Info Register	MBA	ExtRAM
1	I1	32		1	I1	32	I1
2	I2	33		2	I2	33	I2
3	I3	34		3	I3	34	I3
4	I4	35		4	I4	35	I4
5	I5	36		5	I5	36	I5
6	I6	37		6	I6	37	I6
7	I7	38		7	I7	38	I7
8	I8	39		8	I8	39	I8

The STI example stores all eight of the VSP informational registers to external memory beginning at physical address 32. Because all eight registers are stored, STR must be programmed to 1 to begin storage with the accumulators, and NMPT must be programmed to 8 to store all eight registers.

6.3.5 STB (Store Backward)

STB moves data from the internal VSP RAM to external memory in a manner similar to that of the ST instruction. However, with STB the memory base address is *decremented*, not incremented as with ST. In other respects STB is similar to ST.

STB is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 0 | 1 |              NMPT             |RS |       |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   MBS     |   MSS     | RV    |AD | MDF   |   | 1 |           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                             MBAB                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters for the STB instruction are defined in section 6.2: NMPT, RS, EI, MBS, MSS, RV and MDF.

Parameters unique to the STB instruction:

MBAB - Memory Base Address Backward: the base address to use when storing backward. This address will store the imaginary part of the first data point.

Logical N translates to the *ones complement* for N literal.

The following is an example of how the STB instruction may be used:

STB NMPT:8 RS:0 MDF:3 EI:0 MBS:2 MSS:4 RV:0 AD:0 MBAB:41

Before the STB instruction		After the STB instruction	
VSPRAM [0] =	[R0, I0]	ExtRAM [20] =	[R0, I0]
VSPRAM [1] =	[R1, I1]	ExtRAM [19] =	[R1, I1]
VSPRAM [2] =	[R2, I2]	ExtRAM [16] =	[R2, I2]
VSPRAM [3] =	[R3, I3]	ExtRAM [15] =	[R3, I3]
VSPRAM [4] =	[R4, I4]	ExtRAM [12] =	[R4, I4]
VSPRAM [5] =	[R5, I5]	ExtRAM [11] =	[R5, I5]
VSPRAM [6] =	[R6, I6]	ExtRAM [8] =	[R6, I6]
VSPRAM [7] =	[R7, I7]	ExtRAM [7] =	[R7, I7]

Address	VSPRAM	
0	R0	I0
1	R1	I1
2	R2	I2
3	R3	I3
4	R4	I4
5	R5	I5
6	R6	I6
7	R7	I7

Address	ExtRAM
41	I0
40	R0
39	I1
38	R1
37	
36	
35	
34	
33	I2
32	R2
31	I3
30	R3
29	
28	
27	
26	
25	I4
24	R4
23	I5
22	R5
21	
20	
19	
18	
17	I6
16	R6
15	I7
14	R7

The STB example stores eight complex words existing in internal VSP RAM to external memory beginning at physical address 41. Note that the physical address is decremented from address 41. In addition, the MSS and MBS parameters operate in the same manner as they do for the normal store instruction.

6.4 ALU/External Memory Instructions

This section covers the four ALU instructions which operate on two data vectors, one of which must already exist in the internal VSP memory, and the other which must exist in external memory. When these instructions are used, it is not possible for concurrent ALU and I/O operations to be performed because these instructions require the use of both the BIU and the EU. All instructions in this section are three words in length.

The instructions covered in this chapter are:

ADDR (Vector Add Real)
ADDC (Vector Add Complex)
MLTR (Vector Multiply Real Accumulate)
MLTC (Vector Multiply Complex Accumulate)

6.4.1 ADDR (Vector Add Real)

ADDR adds a real vector in external memory to both the real and imaginary parts of a complex vector in internal RAM, and stores the result in internal RAM. External memory remains unchanged. The sum may be stored in internal RAM in one of several forms, as shown below.

ADDR is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 1 | 0 | 1 |           NMPT            |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|    MBS    |    MSS    |       |AD | 1 | 0 |CN |SH |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the ADDR instruction are defined in section 6.2.

The following is an example of how the ADDR instruction may be used. In order to keep the tabular structure used in previous examples, two abbreviations are used below: E --> ExtRAM; V --> VSPRAM

ADDR NMPT:4 RS:0 ADF:3 EI:0 MBS:4 MSS:4 AD:0 CN:0 SH:1 MBA:64

```
    Before Instruction                After Instruction

E[32]   = [D64]; V[0] = [R0,I0]    V[0] = [(D64+R0)/2, (D64+I0)/2]
E[32.1] = [D65]; V[1] = [R1,I1]    V[1] = [(D65+R1)/2, (D65+I1)/2]
E[33]   = [D66]; V[2] = [R2,I2]    V[2] = [(D66+R2)/2, (D66+I2)/2]
E[33.1] = [D67]; V[3] = [R3,I3]    V[3] = [(D67+R3)/2, (D67+I3)/2]

Address   ExtRAM                   Address       VSPRAM
          ______                               ______________________________
  64     |_D64__|                  0     |_(D64+R0)/2_|_(D64+I0)/2_|
  65     |_D65__|                  1     |_(D65+R1)/2_|_(D65+I1)/2_|
  66     |_D66__|                  2     |_(D66+R2)/2_|_(D66+I2)/2_|
  67     |_D67__|                  3     |_(D67+R3)/2_|_(D67+I3)/2_|

            VSPRAM
         ____________
   0    |_R0__|_I0__|
   1    |_R1__|_I1__|
   2    |_R2__|_I2__|
   3    |_R3__|_I3__|
```

The ADDR example adds a four-element real vector beginning at external physical address 64 to both the real and imaginary portions of internal VSP RAM section 0. The SH parameter causes each result to be scaled by two to prevent overflow.

6.4.2 ADDC (Vector Add Complex)

ADDC adds a complex vector in external memory to a complex vector in internal RAM by adding real parts to real parts and imaginary parts to imaginary parts, then stores the sum in internal RAM. External memory remains unchanged.

ADDC is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 1 | 0 | 0 |            NMPT           |RS | 1 | 1 |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|    MBS    |    MSS    |       |AD | 1 | 1 |CN |SH |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the ADDC instruction are defined in section 6.2.

The following is an example of how the ADDC instruction may be used.

ADDC NMPT:4 RS:0 ADF:3 EI:0 MBS:8 MSS:8 AD:0 CN:0 SH:0 MBA:64

```
    Before Instruction                         After Instruction

E[32] = [D64,D65];  V[0] = [R0,I0]        V[0] = [D64+R0, D65+I0]
E[33] = [D66,D67];  V[1] = [R1,I1]        V[1] = [D66+R1, D67+I1]
E[34] = [D68,D69];  V[2] = [R2,I2]        V[2] = [D68+R2, D69+I2]
E[35] = [D70,D71];  V[3] = [R3,I3]        V[3] = [D70+R3, D71+I3]

Address   ExtRAM                     Address      VSPRAM
           _____                              ___________________
  64      |_D64_|                      0     |_D64+R0_|_D65+I0_|
  65      |_D65_|                      1     |_D66+R1_|_D67+I1_|
  66      |_D66_|                      2     |_D68+R2_|_D69+I2_|
  67      |_D67_|                      3     |_D70+R3_|_D71+I3_|
  68      |_D68_|
  69      |_D69_|
  70      |_D70_|
  71      |_D71_|

           VSPRAM
          __________
  0      |_R0_|_I0_|
  1      |_R1_|_I1_|
  2      |_R2_|_I2_|
  3      |_R3_|_I3_|
```

The ADDC example adds a four-element complex vector existing in external RAM beginning at physical address 64 to a complex vector existing inside the VSP in RAM section 0. External memory is unaffected by execution of this instruction.

6.4.3 MLTR (Vector Multiply Real Accumulate)

MLTR multiplies a complex vector in internal RAM by a real vector in external memory. External memory remains unchanged. The product is stored in internal RAM and added to the values in the real and imaginary accumulators. The form of the product is selected as shown below.

MLTR is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 1 | 1 |             NMPT              |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|    MBS    |    MSS    |       |AD | 1 | 0 |CN | 0 |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the MLTR instruction are defined in section 6.2.

The following is an example of how the MLTR instruction may be used.

MLTR NMPT:4 RS:0 ADF:3 EI:0 MBS:4 MSS:4 AD:0 CN:0 MBA:2

```
  Before Instruction                         After Instruction

  ExtRAM[1]    = [D2]              VSPRAM[0] = [R0*D2,  I0*D2]
  ExtRAM[1.1]  = [D3]              VSPRAM[1] = [R1*D3,  I1*D3]
  ExtRAM[2]    = [D4]              VSPRAM[2] = [R2*D4,  I2*D4]
  ExtRAM[2.1]  = [D5]              VSPRAM[3] = [R3*D5,  I3*D5]

   VSPRAM[0] = [R0,I0]         VSPACCUM =
   VSPRAM[1] = [R1,I1]       [ R0*D2 + R1*D3 + R2*D4 + R3*D5,
   VSPRAM[2] = [R2,I2]         I0*D2 + I1*D3 + I2*D4 + I3*D5 ]
   VSPRAM[3] = [R3,I3]

Address    ExtRAM                    Address      VSPRAM
             _____                             _______________
  2        |__D2__|                    0     |_R0*D2_|_I0*D2_|
  3        |__D3__|                    1     |_R1*D3_|_I1*D3_|
  4        |__D4__|                    2     |_R2*D4_|_I2*D4_|
  5        |__D5__|                    3     |_R3*D5_|_I3*D5_|

              VSPRAM
            ___________
  0       |_R0_|__I0_|
  1       |_R1_|__I1_|
  2       |_R2_|__I2_|
  3       |_R3_|__I3_|

REALACCUM -> Rinit                         REALACCUM =
                                 R0*D2 + R1*D3 + R2*D4 + R3*D5

IMAGACCUM -> Rinit                         IMAGACCUM =
                                 I0*D2 + I1*D3 + I2*D4 + I3*D5
```

The MLTR example multiplies a four-element real vector beginning in external memory at physical address 2, with both the real and imaginary vectors existing in the VSP RAM section 0. In addition, at the end of the instruction execution, the real and imaginary accumulators each contain the sum of products obtained during the multiplication process with the respective vector. External memory remains unchanged after the instruction execution.

6.4.4 MLTC (Vector Multiply Complex Accumulate)

MLTC multiplies a complex vector in internal RAM by a complex vector in external memory. External memory remains unchanged. The product is stored in internal RAM, and the sum of the products is stored in the real and imaginary accumulators. The form of the product is selected as shown below.

MLTC is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 1 | 0 |              NMPT              |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   MBS     |    MSS    |       |AD | 1 | 1 |CN |SH |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the MLTC instruction are defined in section 6.2.

The following is an example of how the MLTC instruction may be used.

MLTC NMPT:2 RS:0 ADF:3 EI:0 MBS:2 MSS:2 AD:0 CN:0 SH:0 MBA:0

```
 Before Instruction          After Instruction

ExtRAM[0] = [D0,D1]    VSPRAM[0] = [R0*D0 - I0*D1, I0*D0 + R0*D1]
ExtRAM[1] = [D2,D3]    VSPRAM[1] = [R1*D2 - I1*D3, I1*D2 + R1*D3]

VSPRAM[0] = [R0,I0]        VSPACCUM =
VSPRAM[1] = [R1,I1]           [ R0*D0 - I0*D1 + R1*D2 - I1*D3,
                                I0*D0 + R0*D1 + I1*D2 + R1*D3 ]

Address   ExtRAM              Address           VSPRAM
                                         _________________________________
   0      |__D0__|              0      |_R0*D0-I0*D1_|_I0*D0+R0*D1_|
   1      |__D1__|              1      |_R1*D2-I1*D3_|_I1*D2+R1*D3_|
   2      |__D2__|
   3      |__D3__|

              VSPRAM
          ______________
   0     |__R0__|__I0__|
   1     |__R1__|__I1__|

REALACCUM -> Rinit                   REALACCUM =
                              R0*D0 - I0*D1 + R1*D2 - I1*D3

IMAGACCUM -> Rinit                   IMAGACCUM =
                              I0*D0 + R0*D1 + I1*D2 + R1*D3
```

The MLTC example multiplies two complex vectors existing in external memory beginning at physical address 0 with two complex vectors existing in VSP internal RAM section 0. The real portions of VSP memory contain the real results of the vector multiplies, and the imaginary portions contain the imaginary portions of the vector multiplies. The real accumulator contains the summation of the real components, and the imaginary accumulator contains the summation of the imaginary components. External memory is left unaffected.

6.5 Internal ALU Instructions

There are nine instructions which carry out arithmetic operations within the VSP using its internal registers and memory. Because the *BIU* is not used when these instructions are executed, they may be executed concurrently with I/O instructions. Instructions in this section vary in length from one to three words.

The instructions covered in this section are:

ACCR (Accumulate Real)
ACCI (Accumulate Imaginary)
ABS (Absolute Value)
CMLT (Cross Multiply)
CMCN (Complex Conjugate)
MGSQ (Magnitude Square)
DEMO (Demodulate)
MODLT (Modulate)
SCL (Scale)
SCLT (Scale Literal)

6.5.1 ACCR (Accumulate Real)

ACCR accumulates the real part of the internal vector and stores the result in the real accumulator. Internal memory is not changed.

ACCR is a one-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 1 | 0 | 1 | 0 | 0 |             NMPT              |RS | 0 | 0 |EI |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters have definitions which are the same as those given in section 6.2: NMPT, RS and EI.

The following is an example of how the ACCR instruction may be used.

ACCR NMPT:4 RS:0 EI:0

```
    Before Instruction                    After Instruction

VSPRAM[0] = [ R0, I0 ]              VSPRAM[0] = [ R0, I0 ]
VSPRAM[1] = [ R1, I1 ]              VSPRAM[1] = [ R1, I1 ]
VSPRAM[2] = [ R2, I2 ]              VSPRAM[2] = [ R2, I2 ]
VSPRAM[3] = [ R3, I3 ]              VSPRAM[3] = [ R3, I3 ]
```

VSPACCUM = [Rinit,Iinit] VSPACCUM = $[\sum_{i=0}^{NMPT-1} R_i, \text{Iinit}]$

```
Address      VSPRAM            Address      VSPRAM

  0      |__R0__|__I0__|         0      |__R0__|__I0__|
  1      |__R1__|__I1__|         1      |__R1__|__I1__|
  2      |__R2__|__I2__|         2      |__R2__|__I2__|
  3      |__R3__|__I3__|         3      |__R3__|__I3__|
```

REALACCUM -> Rinit REALACCUM = $\sum_{i=0}^{NMPT-1} R_i$

IMAGACCUM -> Iinit IMAGACCUM -> Iinit

The ACCR example performs an accumulation on the real portion of a four-element vector existing in VSP RAM section 0. The result is left in the real accumulator. Neither the imaginary accumulator nor the internal memory are affected by the ACCR instruction.

6.5.2 ACCI (Accumulate Imaginary)

ACCI accumulates the imaginary part of the internal vector and stores the result in the imaginary accumulator. Internal memory is not changed.

ACCI is a one-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 1 | 1 | 1 | 0 | 1 |           NMPT            |RS | 0 | 0 |EI |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters have definitions which are the same as those given in section 6.2: NMPT, RS and EI.

The following is an example of how the ACCI instruction may be used.

ACCI NMPT:4 RS:0 EI:0

```
   Before Instruction                 After Instruction

VSPRAM[0] = [ R0, I0 ]            VSPRAM[0] = [ R0, I0 ]
VSPRAM[1] = [ R1, I1 ]            VSPRAM[1] = [ R1, I1 ]
VSPRAM[2] = [ R2, I2 ]            VSPRAM[2] = [ R2, I2 ]
VSPRAM[3] = [ R3, I3 ]            VSPRAM[3] = [ R3, I3 ]
```

VSPACCUM = [Rinit, Iinit]   VSPACCUM =[Rinit, $\sum_{i=0}^{NMPT-1} I_i$]

```
Address      VSPRAM           Address      VSPRAM

   0     |__R0__|__I0__|         0     |__R0__|__I0__|
   1     |__R1__|__I1__|         1     |__R1__|__I1__|
   2     |__R2__|__I2__|         2     |__R2__|__I2__|
   3     |__R3__|__I3__|         3     |__R3__|__I3__|
```

REALACCUM -> Rinit   REALACCUM -> Rinit

IMAGACCUM -> Iinit   IMAGACCUM = $\sum_{i=0}^{NMPT-1} I_i$

The ACCI example performs an accumulation on the imaginary portion of a four-element vector existing in VSP RAM section 0. The result is left in the imaginary accumulator. Neither the real accumulator nor the internal memory are affected by the ACCI instruction.

6.5.3 ABS (Absolute Value)

ABS causes selected parts of the internal vector to be replaced by their absolute values. ADF specifies whether only the real part, only the imaginary part, or both parts will be replaced.

ABS is a one-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 1 | 1 | 1 | 1 | 1 |            NMPT            |RS |  ADF  |EI |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the ABS instruction are defined in section 6.2.

The following is an example of how the ABS instruction may be used.

ABS NMPT:4 RS:1 ADF:3 EI:0

```
   Before Instruction                   After Instruction

VSPRAM[64] = [ R64, I64 ]          VSPRAM[64] = [ |R64|, |I64| ]
VSPRAM[65] = [ R65, I65 ]          VSPRAM[65] = [ |R65|, |I65| ]
VSPRAM[66] = [ R66, I66 ]          VSPRAM[66] = [ |R66|, |I66| ]
VSPRAM[67] = [ R67, I67 ]          VSPRAM[67] = [ |R67|, |I67| ]
```

```
Address       VSPRAM               Address        VSPRAM

   0     |  R64  |  I64  |           64    |  |R64|  |  |I64|  |
   1     |  R65  |  I65  |           64    |  |R65|  |  |I65|  |
   2     |  R66  |  I66  |           64    |  |R66|  |  |I66|  |
   3     |  R67  |  I67  |           64    |  |R67|  |  |I67|  |
```

The ABS example takes the absolute value of both the real and imaginary parts of a four-element complex vector stored inside the VSP in RAM section 1.

6.5.4 CMLT (Cross Multiply Accumulate)

CMLT multiplies the real part of the internal vector by the imaginary part, then stores the result in the real part. The summation of these products also goes to the real accumulator. ADF must be 10 or 00. If ADF is 00, only the real accumulator is changed.

CMLT is a one-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 1 | 1 | 1 | 1 | 1 |              NMPT               |RS |  ADF  |EI |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the CMLT instruction are defined in section 6.2.

The following is an example of how the CMLT instruction may be used.

CMLT NMPT:6 RS:1 ADF:2 EI:0

<u>**Before Instruction**</u>	<u>**After Instruction**</u>
VSPRAM[64] = [R64, I64]	VSPRAM[64] = [R64*I64, I64]
VSPRAM[65] = [R65, I65]	VSPRAM[65] = [R65*I65, I65]
VSPRAM[66] = [R66, I66]	VSPRAM[66] = [R66*I66, I66]
VSPRAM[67] = [R67, I67]	VSPRAM[67] = [R67*I67, I67]
VSPRAM[67] = [R68, I68]	VSPRAM[68] = [R68*I68, I68]
VSPRAM[67] = [R69, I69]	VSPRAM[69] = [R69*I69, I69]
VSPACCUM = [Rinit, Iinit]	VSPACCUM = $[\sum_{i=0}^{NMPT-1} R_i * I_i, Iinit]$

Address	**VSPRAM**		**Address**	**VSPRAM**	
64	R64	I64	64	R64*I64	I64
65	R65	I65	65	R65*I65	I65
66	R66	I66	66	R66*I66	I66
67	R67	I67	67	R67*I67	I67
68	R68	I68	68	R68*I68	I68
69	R69	I69	69	R69*I69	I69

REALACCUM -> Rinit	REALACCUM = $\sum_{i=0}^{NMPT-1} (R_i * I_i)$
IMAGACCUM -> Iinit	IMAGACCUM -> Iinit

6.5.5 CMCN (Complex Conjugate)

CMCN replaces the complex internal vector with its complex conjugate. ADF must be 11 binary.

CMCN is a one-word instruction. CMCN may be used as a NOP instruction by setting ADF=0 and setting NMPT=1.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 1 | 1 | 0 | 1 |               NMPT                |RS |  ADF  |EI |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the CMCN instruction are defined in section 6.2.

The following is an example of how the CMCN instruction may be used.

CMCN NMPT:5 RS:0 ADF:3 EI:0

```
   Before Instruction                    After Instruction

VSPRAM[0] = [ R0, I0 ]            VSPRAM[0] = [ R0, -I0 ]
VSPRAM[1] = [ R1, I1 ]            VSPRAM[1] = [ R1, -I1 ]
VSPRAM[2] = [ R2, I2 ]            VSPRAM[2] = [ R2, -I2 ]
VSPRAM[3] = [ R3, I3 ]            VSPRAM[3] = [ R3, -I3 ]
VSPRAM[4] = [ R4, I4 ]            VSPRAM[4] = [ R4, -I4 ]

Address      VSPRAM               Address      VSPRAM

   0     |__R0__|__I0__|             0     |__R0__|__-I0__|
   1     |__R1__|__I1__|             1     |__R1__|__-I1__|
   2     |__R2__|__I2__|             2     |__R2__|__-I2__|
   3     |__R3__|__I3__|             3     |__R3__|__-I3__|
   4     |__R4__|__I4__|             4     |__R4__|__-I4__|
```

6.5.6 MGSQ (Magnitude Square Accumulate)

MGSQ calculates the square of the magnitude of the internal vector. The result is scaled down by two to prevent overflow, and is written into the real part of the internal memory. The sum of the magnitude squared elements is stored in the real accumulator. ADF must be 10 or 00. If ADF is 00, only the accumulators are updated.

MGSQ is a one-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 1 | 0 | 1 | 1 | 1 |           NMPT                |RS |  ADF  |EI |
+----------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All of the parameters contained in the MGSQ instruction are defined in section 6.2.

The following is an example of how the MGSQ instruction may be used.

MGSQ NMPT:4 RS:0 ADF:2

<u>Before Instruction</u>	<u>After Instruction</u>
VSPRAM[0] = [R0, I0]	VSPRAM[0] = [$(R0^2 + I0^2)/2$, I0]
VSPRAM[1] = [R1, I1]	VSPRAM[1] = [$(R1^2 + I1^2)/2$, I1]
VSPRAM[2] = [R2, I2]	VSPRAM[2] = [$(R2^2 + I2^2)/2$, I2]
VSPRAM[3] = [R3, I3]	VSPRAM[3] = [$(R3^2 + I3^2)/2$, I3]

$$\text{VSPACCUM} = [\ 1/2 * \sum_{i=0}^{\text{NMPT}-1} (R_i^2 + I_i^2),\ \text{Iinit}\ ]$$

Address	VSPRAM		Address	VSPRAM	
0	R0	I0	0	$(R0^2 + I0^2)/2$	I0
1	R1	I1	1	$(R1^2 + I1^2)/2$	I1
2	R2	I2	2	$(R2^2 + I2^2)/2$	I2
3	R3	I3	3	$(R3^2 + I3^2)/2$	I3

REALACCUM -> Rinit	REALACCUM = $1/2\sum_{i=0}^{\text{NMPT}-1} (R_i^2 + I_i^2)$
IMAGACCUM -> Iinit	IMAGACCUM -> Iinit

6.5.7 DEMO (Demodulate)

DEMO multiplies a complex vector in internal memory by a series of complex coefficients generated from the sine-cosine look-up table. The coefficients are specified in the instruction by a ROM base address and decrement address. The result is stored in the real and imaginary accumulators.

The multiply operation follows the formula:

VSPRAM [i] = VSPRAM [i] * (cos O - j sin O)
where O = RBA + RDA * i

The VSPROM contains 256 cosine values from 0 to PI/2. The multiplication performed in the DEMO instruction demodulates or frequency translates an input signal by performing an element-by-element complex multiplication with a complex sinusoid. RBA corresponds to the initial phase and RDA determines the frequency of the sinusoid. DEMO is precisely the same as the MODLT instruction in section 6.5.8 except for the definition of the complex sinusoid. The imaginary portion of the sinusoid is decremented instead of incremented as in the MODLT instruction.

DEMO is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 1 | 0 | 1 | 0 |             NMPT              |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 |                 RDA                   |       |SH |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                   RBA                     | 1 |           | 1 | 1 |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters for the DEMO instruction are defined in section 6.2: NMPT, RS, ADF, EI and SH.

Parameters unique to the DEMO instruction:

RDA - ROM Decrement Address: address used to define the incremental angles for successive coefficients.

DEGREES*10 is the logical value for RDA.
DEGREES*1024/360 is the literal value for RDA.

RBA - ROM Base Address: address of the start of the cosine values for the 1024 angles from 0 to 2 PI.

DEGREES*10 is the logical value for RBA.
DEGREES*1024/360 is the literal value for RBA.

FSIZ - Specifies the number of samples beginning with RBA to be addressed from the internal sine/cosine LUT, after which the LUT address rolls back to the RBA value.

Literal		Logical
000	->	8 points
001	->	16 points
010	->	32 points
011	->	64 points
100	->	128 points

Logical 2^(N+3) translates to N literal.

The following is an example of how the DEMO instruction may be used.

DEMO NMPT:4 RS:0 ADF:3 EI:0 RDA:900 RBA:450 SH:0

```
          Before Instruction

VSPACCUM = [Rinit,Iinit]

VSPRAM[0] = [R0, I0]          VSPROM[128] = [ .707, -.707]
VSPRAM[1] = [R1, I1]          VSPROM[384] = [-.707, -.707]
VSPRAM[2] = [R2, I2]          VSPROM[640] = [-.707,  .707]
VSPRAM[3] = [R3, I3]          VSPROM[896] = [ .707,  .707]

          After Instruction

VSPRAM[0] = [ .707(I0 + R0),  .707(I0 - R0) ]
VSPRAM[1] = [ .707(I1 - R1), -.707(R1 + I1) ]
VSPRAM[2] = [-.707(I2 + R2),  .707(R2 - I2) ]
VSPRAM[3] = [ .707(R3 - I3),  .707(R3 + I3) ]

VSPACCUM = [ .707(R0-R1-R2+R3+I0+I1-I2-I3),
             .707(-R0-R1+R2+R3+I0-I1-I2+I3) ]

   Before Instruction             After Instruction

Address      VSPRAM           Address            VSPRAM

   0      |  R0  |  I0  |        0      |  .707(I0+R0)  |  .707(I0-R0)  |
   1      |  R1  |  I1  |        1      |  .707(I1-R1)  | -.707(R1+I1)  |
   2      |  R2  |  I2  |        2      | -.707(I2+R2)  |  .707(R2-I2)  |
   3      |  R3  |  I3  |        3      |  .707(R3-I3)  |  .707(I3+R3)  |

REALACCUM -> Rinit      REALACCUM =  .707(R0-R1-R2+R3+I0+I1-I2-I3)

IMAGACCUM -> Iinit      IMAGACCUM = .707(-R0-R1+R2+R3+I0-I1-I2+I3)
```

The DEMO example multiplies a four-element complex vector with a complex sinusoid, the result of which is equivalent to a demodulation of the complex vector. The real results of the demodulation are left in the real portion of memory, while the imaginary results are left in the imaginary portion of memory. The real accumulator contains the accumulation of the real portion of the demodulation. The imaginary accumulator contains the accumulation of the imaginary portion of the demodulation. The RBA parameter specifies the phase offset or initial phase angle; 45^o in this example. The RDA parameter specifies the phase increment (frequency); 90^o in this example.

6.5.8 MODLT (Modulate)

MODLT multiplies a complex vector in internal memory by a series of complex coefficients generated from the sine-cosine look-up table. The coefficients are specified in the instruction by a ROM base address and increment address. The result is stored in the real and imaginary accumulators.

The multiply operation follows the formula:

$$\text{VSPRAM}[i] = \text{VSPRAM}[i] * (\cos O + j \sin O)$$
$$\text{where } O = \text{RBA} + \text{RIA} * i$$

The VSPROM contains 256 cosine values from 0 to PI/2. The multiplication performed in the MODLT instruction modulates or frequency translates an input signal by performing an element-by-element complex multiplication with a complex sinusoid. RBA corresponds to the initial phase and RIA determines the frequency of the sinusoid. MODLT is precisely the same as the DEMO instruction in section 6.5.7 except for the definition of the complex sinusoid. The imaginary portion of the sinusoid is incremented instead of decremented as in the DEMO instruction.

MODLT is a three-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 1 | 0 | 1 | 0 |           NMPT                |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 |              RIA                  |       |SH |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                  RBA                  | FSIZ      |   | 1 | 1 |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters for the MODLT instruction are defined in section 6.2: NMPT, RS, ADF, EI and SH.

Parameters unique to the MODLT instruction:

RIA - ROM Increment Address: address used to define the incremental angles for successive coefficients.

DEGREES*10 is the logical value for RIA.
DEGREES*1024/360 is the literal value for RIA.

RBA - ROM Base Address: address of the start of the cosine values for the 1024 angles from 0 to 2 PI.

DEGREES*10 is the logical value for RBA.
DEGREES*1024/360 is the literal value for RBA.

FSIZ - Specifies the number of samples beginning with RBA to be addressed from the internal sine/cosine LUT, after which the LUT address rolls back to the RBA value.

Literal		**Logical**
000	->	8 points
001	->	16 points
010	->	32 points
011	->	64 points
100	->	128 points

Logical 2^(N+3) translates to N literal.

The following is an example of how the MODLT instruction may be used.

MODLT NMPT:4 RS:0 ADF:3 EI:0 RIA:900 RBA:450 SH:0

```
            Before Instruction

VSPACCUM = [Rinit,Iinit]

VSPRAM[0] = [R0, I0]          VSPROM[128] = [ .707,   .707]
VSPRAM[1] = [R1, I1]          VSPROM[384] = [-.707,   .707]
VSPRAM[2] = [R2, I2]          VSPROM[640] = [-.707, -.707]
VSPRAM[3] = [R3, I3]          VSPROM[896] = [ .707, -.707]

            After Instruction

VSPRAM[0] = [ .707(R0 - I0),   .707(R0 + I0) ]
VSPRAM[1] = [-.707(R1 + I1),   .707(R1 - I1) ]
VSPRAM[2] = [ .707(I2 - R2), -.707(R2 + I2) ]
VSPRAM[3] = [ .707(R3 + I3),   .707(I3 - R3) ]

VSPACCUM = [ .707(R0-R1-R2+R3-I0-I1+I2+I3),
             .707(R0+R1-R2-R3+I0-I1-I2+I3) ]
```

```
    Before Instruction          After Instruction

Address     VSPRAM          Address           VSPRAM

   0     |__R0__|__I0__|       0     |__.707(R0-I0)_|__.707(R0+I0)_|
   1     |__R1__|__I1__|       1     |_-.707(R1+I1)_|__.707(R1-I1)_|
   2     |__R2__|__I2__|       2     |__.707(I2-R2)_|_-.707(R2+I2)_|
   3     |__R3__|__I3__|       3     |__.707(R3+I3)_|__.707(I3-R3)_|

REALACCUM -> Rinit      REALACCUM =  .707(R0-R1-R2+R3-I0-I1+I2+I3)

IMAGACCUM -> Iinit      IMAGACCUM =  .707(R0+R1-R2-R3+I0-I1-I2+I3)
```

The MODLT example multiplies a four-element complex vector with a complex sinusoid, the result of which is equivalent to a modulation of the complex vector.

The real results of the modulation are left in the real portion of memory, while the imaginary results are left in the imaginary portion of memory. The real accumulator contains the accumulation of the real portion of the modulation. The imaginary accumulator contains the accumulation of the imaginary portion of the modulation. The RBA parameter specifies the phase offset; 45^o in this example. The RIA parameter specifies the phase increment (frequency); 90^o in this example.

6.5.9 SCL (Scale)

SCL scales an internal complex vector down in magnitude by performing an integer number of right shifts on the data samples of the vector operand. The number of bits of right shifting performed is determined by elements of a scale vector which must have been previously loaded into the VSP scale RAM.

The scale vector must be written by the host into the scale RAM prior to execution of the SCL command, or loaded with the LDSM instruction. The scale vector can be 1 to 64 nibbles in length. Each nibble is a scaling factor from 0 to 15, representing the number of right-shifts -- divide by twos -- to apply to the elements of the internal vector. The simplest example is a scale vector the same length as the internal operand vector, where different parts of the latter are scaled by different factors in the scale vector. If multiple successive points of the internal operand vector are to be scaled by the same scale nibble, the scale vector may be shorter than the operand.

The length of the scale vector is specified in the instruction setup. Also specified is the number of successive points in the internal vector to be scaled by the same scaling factor. When the scale vector length is shorter than the operand length, the scale vector starts over at its beginning to process the remainder of the operand. If the scale vector is only one nibble in length, the instruction allows specification of which nibble out of the first four in the Scale RAM is used in the instruction execution.

SCL also sums the scaled results into both the real and imaginary accumulators.

The content of each nibble of the scale vector is the number of right shifts (divide-by-twos) performed on the operand vector:

0000 -> no effect
0001 -> divide by 2
1111 -> divide by 32,768 (2**15)

SCL is a two-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 1 | 0 | 1 | 1 |           NMPT            |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                   | 0 |SB |   SCLVL   |   SCLBL   |       |LN |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters for the SCL instruction are defined in section 6.2: NMPT, RS, ADF and EI.

Parameters unique to the SCL instruction:

SB - Subtract: source of the content of the scale factor.

0	->	use the factor in scale RAM
1	->	use the old maximum scale value *minus* the scale RAM value

SCLVL - Scale Vector Length

Literal		**Logical**
000	->	1 nibble
110	->	64 nibbles

Logical 2^N is translated to N literal.

SCLBL - Scale Block Length: SCLBL is the number of points in VSP RAM to have the same scale factor.

Literal		**Logical**
000	->	1 point
101	->	32 points

Logical 2^N is translated to N literal.

If the scale vector length is 1, SCLBL defines the nibble in the first four Scale RAM nibbles to use.

LN - word length of instruction:

0	->	used as a three-word instruction
1	->	used as a two-word instruction

The following is an example of how the SCL instruction may be used:

SCL NMPT:10 RS:0 ADF:3 EI:0 SB:1 SCLVL:4 SCLBL:2 LN:1

```
   Before Instruction                         After Instruction

VSPRAM[0] = [ R0, I0 ]               VSPRAM[0] = [ R0/8, I0/8 ]
VSPRAM[1] = [ R1, I1 ]               VSPRAM[1] = [ R1/8, I1/8 ]
VSPRAM[2] = [ R2, I2 ]               VSPRAM[2] = [ R2/4, I2/4 ]
VSPRAM[3] = [ R3, I3 ]               VSPRAM[3] = [ R3/4, I3/4 ]
VSPRAM[4] = [ R4, I4 ]               VSPRAM[4] = [ R4/16, I4/16 ]
VSPRAM[5] = [ R5, I5 ]               VSPRAM[5] = [ R5/16, I5/16 ]
VSPRAM[6] = [ R6, I6 ]               VSPRAM[6] = [ R6/8, I6/8 ]
VSPRAM[7] = [ R7, I7 ]               VSPRAM[7] = [ R7/8, I7/8 ]
VSPRAM[8] = [ R8, I8 ]               VSPRAM[8] = [ R8/8, I8/8 ]
VSPRAM[9] = [ R9, I9 ]               VSPRAM[9] = [ R9/8, I9/8 ]

SCLRAM[0] = [ 2312H ]
OMSCLRAM  = [ 5 ]

Address       VSPRAM          Address        VSPRAM

   0     |__R0__|__I0__|         0     |__R0/8__|__I0/8__|
   1     |__R1__|__I1__|         1     |__R1/8__|__I1/8__|
   2     |__R2__|__I2__|         2     |__R2/4__|__I2/4__|
   3     |__R3__|__I3__|         3     |__R3/4__|__I3/4__|
   4     |__R4__|__I4__|         4     |__R4/16_|__I4/16_|
   5     |__R5__|__I5__|         5     |__R5/16_|__I5/16_|
   6     |__R6__|__I6__|         6     |__R6/8__|__I6/8__|
   7     |__R7__|__I7__|         7     |__R7/8__|__I7/8__|
   8     |__R8__|__I8__|         8     |__R8/8__|__I8/8__|
   9     |__R9__|__I9__|         9     |__R9/8__|__I9/8__|

              SCLRAM

   0    |__2__|__3__|__1__|__2__|

Old Maximum Scale Register   |__5__|

REALACCUM -> Rinit          REALACCUM = (R0+R1)/8 + (R2+R3)/4 +
                                   (R4+R5)/16 + (R6+R7+R8+R9)/8

IMAGACCUM -> Iinit          IMAGACCUM = (I0+I1)/8 + (I2+I3)/4 +
                                   (I4+I5)/16 + (I6+I7+I8+I9)/8
```

In the SCL example, the scale RAM has been previously loaded with a scale vector with a length of four (SCLVL:4) nibbles. The old maximum scale RAM contains the hexadecimal nibble 5H. The number of bits of right-shift performed on each complex data sample is determined by subtracting the respective scale constant for each sample from the old max scale RAM value (SB:1). Each value in the scale RAM scales two complex samples (SCLBL:2). After the first eight complex samples are scaled, the pointer to the scale RAM rolls over and points to the first value again; VSPRAM[8] and VSPRAM[9] are scaled by the same shift as VSPRAM[0] and VSPRAM[1]. The summation of the ten scaled values is stored in both the real and imaginary accumulators.

6.5.10 SCLT (Scale Literal)

SCLT scales the internal vector by a constant defined by the SHF parameter. SCLT is a two-word instruction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 1 | 0 | 1 | 1 |           NMPT                |RS |  ADF  |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                   | 1 |SB |       |       SHF     |       |LN |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters for the SCLT instruction are defined in section 6.2: NMPT, RS, ADF and EI.

Parameters unique to the SCLT instruction:

SHF - Shift: the number of right-shifts to apply to each vector element.

<u>Literal</u>		<u>Logical</u>
0000	->	no effect
0001	->	divide by two
1111	->	divide by 2**15, or 32,768

SB - Subtract: Source of the content of the scale factor.

0	->	use the SHF parameter as defined in the instruction for the number of right-shifts
1	->	use the old scale RAM value minus the SHF value as the number of right-shifts to perform

The following is an example of how the SCLT instruction may be used:

SCLT NMPT:5 RS:0 ADF:3 EI:0 SB:1 SHF:2 LN:1

```
    Before Instruction                   After Instruction

VSPRAM[0] = [ R0, I0 ]            VSPRAM[0] = [ R0/8, I0/8 ]
VSPRAM[1] = [ R1, I1 ]            VSPRAM[1] = [ R1/8, I1/8 ]
VSPRAM[2] = [ R2, I2 ]            VSPRAM[2] = [ R2/8, I2/8 ]
VSPRAM[3] = [ R3, I3 ]            VSPRAM[3] = [ R3/8, I3/8 ]
VSPRAM[4] = [ R4, I4 ]            VSPRAM[4] = [ R4/8, I4/8 ]

OMSCLRAM  = [ 5 ]

Address     VSPRAM           Address      VSPRAM
          ____________                 ______________
   0     |_R0_|_I0_|            0     |_R0/8_|_I0/8_|
   1     |_R1_|_I1_|            1     |_R1/8_|_I1/8_|
   2     |_R2_|_I2_|            2     |_R2/8_|_I2/8_|
   3     |_R3_|_I3_|            3     |_R3/8_|_I3/8_|
   4     |_R4_|_I4_|            4     |_R4/8_|_I4/8_|

                                 ___
Old Maximum Scale Register      |_5_|
```

The SCLT example scales the five complex samples in VSP RAM section 0 by the difference between the old max scale RAM value (5H) and the shift parameter specified in the instruction field (SHF:2). In this example, all five of the complex samples are shifted right by three.

6.6 Control Instructions

This section covers the three instructions which control program flow in the VSP. They vary in length from one to three words. The three instructions are:

JMPI (Jump Indirect)
HLT (Halt)
NOP (No Operation)

6.6.1 JMPI (Jump Indirect)

JMPI is the main program flow control instruction within the VSP. JMPI causes the VSP to load a new instruction fetch address located at the memory base address defined by MBA in the instruction word. JMPI is executed by the BIU.

JMPI may be used for both calling and returning from subroutines.

JMPI is a three-word instuction.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |RS | 0 | 0 |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 |               | 1 | 0 | 1 | 1 | 1 | 0 |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                              MBA                              |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

6.6.2 HLT (Halt)

HLT stops the VSP bus-interface unit from fetching any more instructions. It is always used as the last instruction in a program or when it is desired to halt instruction fetch. ALU instructions executing or queued in the instruction FIFO when a HLT instruction is executed are not affected. ALU instructions will complete and provide status to the host as defined in the particular ALU instruction. HLT has no meaning in the slave mode.

HLT is a two-word instruction, where all but the first five bits are DON'T CARE.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 1 | 1 | 0 | 0 | 0 |                                           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                                               |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

Examples of the use of JMPI and HLT are presented in Chapter X.

6.6.3 NOP (No Operation)

NOP is a one-word null instruction which has no effect on the execution of other instructions, nor on registers (except the status registers which are updated) nor memory. NOP is implemented as a CMCN instruction with ADF:00. Note that the execution time of the NOP instruction is a function of the NMPT parameter defined in the instruction. This allows variable-length NOP instructions for applications requiring predictable delays. It is sometimes useful to insert NOPs in a program to reserve space for later use or to time operations for real-time applications.

```
       6       |       ?       |       0       | 0           1
+---.---.---.---|---.---.---.---|---.---.---.---|---.---.---.---+
| 0 | 1 | 1 | 0 | 1 |             NMPT            |RS | 0   0 |EI |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

All programmable parameters in the NOP instruction are defined in section 6.2.

6.7 The FFT Instruction

The FFT instruction is used to perform Fast Fourier Transforms on real or complex vectors stored in the VSP internal memory. This chapter covers first the instruction itself including a description of its parameters. Next it explains the uses and inter-relationships of the parameters and how they control the execution of a transform. Finally there is an example of an FFT algorithm which illustrates the material already covered.

6.7.1 FFT - The Fast Fourier Transform Instruction

FFT executes a Fast Fourier Transform or an Inverse Fast Fourier Transform on data stored internally in the VSP RAM. The associated parameters give this instruction a wide range of flexibility.

```
+---.---.---.---.---.---.---.---.---.---.---.---.---.---.---.---+
| 1 | 0 | 0 | 1 | 0 |              NMBT             |RS | 1 | 1 |EI |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| R | 0 | 0 | 0 | 0 | 0 | 0 |     FPS     |     LPS     |       | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 |                 RBA                 |    FSIZ   |AS | I | R |
+---------------------------------------------------------------+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

The following parameters have definitions which are the same as those given in section 6.2: RS and EI.

Parameters which are unique to the FFT instruction:

NMBT - Number of Butterflies per pass

The number of butterflies (NMBT) is a **literal value** describing the number of butterfly operations which are to be performed. Literal NMBT is in the range of 1 to 64.

Logical NMBT is the number of data points, which is twice the number of butterflies. *When using the simulator with logical parameters, NMBT must be thought of as the number of samples, not the number of butterflies.* Logical NMBT is in the range 2 to 128.

R - Reverse: order of the data in internal memory

0 -> normal order
1 -> bit-reversed order

When R=1, FPS should be 6 literal, LPS less than 6 literal, and RBA=0.

Note that R occurs a second time in the instruction parameters as the last parameter in the third word.

FPS - First Pass Separation: the separation of the two sample points in the first butterfly pass.

Literal		Logical
000	->	64-point separation
110	->	1-point separation

Logical 2^(6-N) is translated to N literal.

LPS - Last Pass Separation: the separation of the two sample points in the last butterfly pass

Literal		Logical
000	->	64-point separation
110	->	1-point separation

Logical 2^(6-N) is translated to N literal.

RBA - ROM Base Address: the offset address from 0 to 511 (representing angles from 0 to PI) of the first coefficient to be used in the FFT in each pass. In each successive pass, RBA is right-shifted one bit.

DEGREES*10 is the logical value for RBA.
DEGREES*1024/360 is the literal value for RBA.

FSIZ - FFT Size: the number of points contained in the FFT. Programming FSIZ to a value different (smaller) than NMBT allows multiple FFTs to be computed using a single FFT instruction. The total number of points is the product of the number of FFTs and FSIZ. For example, the partial instruction:

```
FFT NMBT:128, FPS:16, LPS:1, FSIZ:32, RBA:0;
```

will calculate four 32-point FFTs using a single FFT instruction.

Literal		Logical
000	->	8-point FFT
001	->	16-point FFT
010	->	32-point FFT
011	->	64-point FFT
100	->	128-point FFT

Logical 2^(N+3) is translated to N literal.

AS - Automatic Scale: chooses the type of scaling to perform in conjunction with the FFT calculations.

0 -> Block floating operation. Scaling will be performed manually with the scale instruction.
1 -> Fixed divide by two each pass. Normally the scale instruction is not used with AS=1.

Note that it is possible to experience an overflow when AS=1.

I - Inverse

0 -> forward FFT
1 -> inverse FFT

The following is an example of how the FFT instruction may be used.

FFT NMPT:8 RS:0 EI:0 FPS:4 LPS:1 RBA:0 FSIZ:8 AS:1 I:0 R:0

```
   Before Instruction                     After Instruction

VSPRAM[0] = [ R0, I0 ]             VSPRAM[0] = [ FR0, FI0 ]
VSPRAM[1] = [ R1, I1 ]             VSPRAM[1] = [ FR4, FI4 ]
VSPRAM[2] = [ R2, I2 ]             VSPRAM[2] = [ FR2, FI2 ]
VSPRAM[3] = [ R3, I3 ]             VSPRAM[3] = [ FR6, FI6 ]
VSPRAM[4] = [ R4, I4 ]             VSPRAM[4] = [ FR1, FI1 ]
VSPRAM[5] = [ R5, I5 ]             VSPRAM[5] = [ FR5, FI5 ]
VSPRAM[6] = [ R6, I6 ]             VSPRAM[6] = [ FR3, FI3 ]
VSPRAM[7] = [ R7, I7 ]             VSPRAM[7] = [ FR7, FI7 ]
```

```
Address        VSPRAM          Address        VSPRAM
             ____________                  _____________
   0        |__R0__|__I0__|       0        |__FR0__|__FI0__|
   1        |__R1__|__I1__|       1        |__FR4__|__FI4__|
   2        |__R2__|__I2__|       2        |__FR2__|__FI2__|
   3        |__R3__|__I3__|       3        |__FR6__|__FI6__|
   4        |__R4__|__I4__|       4        |__FR1__|__FI1__|
   5        |__R5__|__I5__|       5        |__FR5__|__FI5__|
   6        |__R6__|__I6__|       6        |__FR3__|__FI3__|
   7        |__R7__|__I7__|       7        |__FR7__|__FI7__|
```

FR and FI are the real and imaginary parts of the transform.

The FFT example takes a complex eight-point FFT of data stored in VSP RAM section 0. The results of this FFT are also stored in VSP RAM section 0. The first-pass spacing between data points is four, and the last-pass spacing is one. This is consistent with normally-ordered input data. The results stored in internal VSP memory after the FFT calculation are in bit-reversed order.

6.7.2 The FFT Algorithm

The Fast Fourier Transform and its inverse are mathematically-efficient algorithms for implementation of the the discrete Fourier transform (DFT) and the inverse discrete Fourier transform (IDFT). "Mathematically-efficient" means that the number of multiplications and additions required to complete the FFT calculation are much fewer than the number required for calculation of the DFT. The mathematical definitions of the DFT and IDFT are shown below. The number of multiplications and additions required for the computation of the *discrete Fourier transform* is *of the order of* N^2. The number of calculations required for computation of the FFT algorithm is *of the order of* $N \log_2 (N)$. Because of the mathematical efficiency of the FFT algorithm, it is a widely used technique.

The DFT is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(2*pi*k*n/N)}$$

The IDFT is defined as:

$$x(n) = 1/N \sum_{k=0}^{N-1} X(k)e^{+j(2*pi*k*n/N)}$$

where

$x(n)$ is a time-domain sequence of length N samples,

and

$X(k)$ is the transform of $x(n)$.

Oftentimes the exponential factor $e^{-j(2*pi*k*n/N)}$ is simplified to the form below:

$$W_N^k = e^{-j(2*pi*k*n/N)},$$

where k and N are as defined above.

The FFT instruction in the VSP is a very powerful command which allows a great deal of flexibility. This flexibility allows calculations of transforms of different lengths and dimensions. For shorter length transforms, multiple FFTs may be calculated simultaneously. For instance, when calculating a 16x16 transform, multiple rows of the two-dimensional transform may be calculated simultaneously. The parameters of the instruction control how the algorithm is implemented. The following sections describe the FFT algorithm and the relationship between the instruction and the algorithm.

6.7.3 The Decimation-in-Time Algorithm

Many efficient mathematical techniques and algorithms have been developed over the last several years for computing the FFT. The decimation in time (DIT) algorithm is one commonly used. It is shown in flowchart forms in Figures 6-2 and 6-3. Figure 6-2 shows how the DIT algorithm begins computation of the FFT by forming small sub-sequences of the input time sequence and performing small transforms on *butterflies*. Successively larger transforms are formed from the smaller ones until the complete transform of the desired length is achieved. An expanded view of the signal processing involved in each butterfly is shown in figure 6-4.

The VSP FFT instruction executes an FFT and an inverse FFT using the DIT algorithm. All the principles and rules in the following sections for the FFT are also applicable to the inverse FFT. The properties of the DIT algorithm are described and related to the parameters of the FFT instruction.

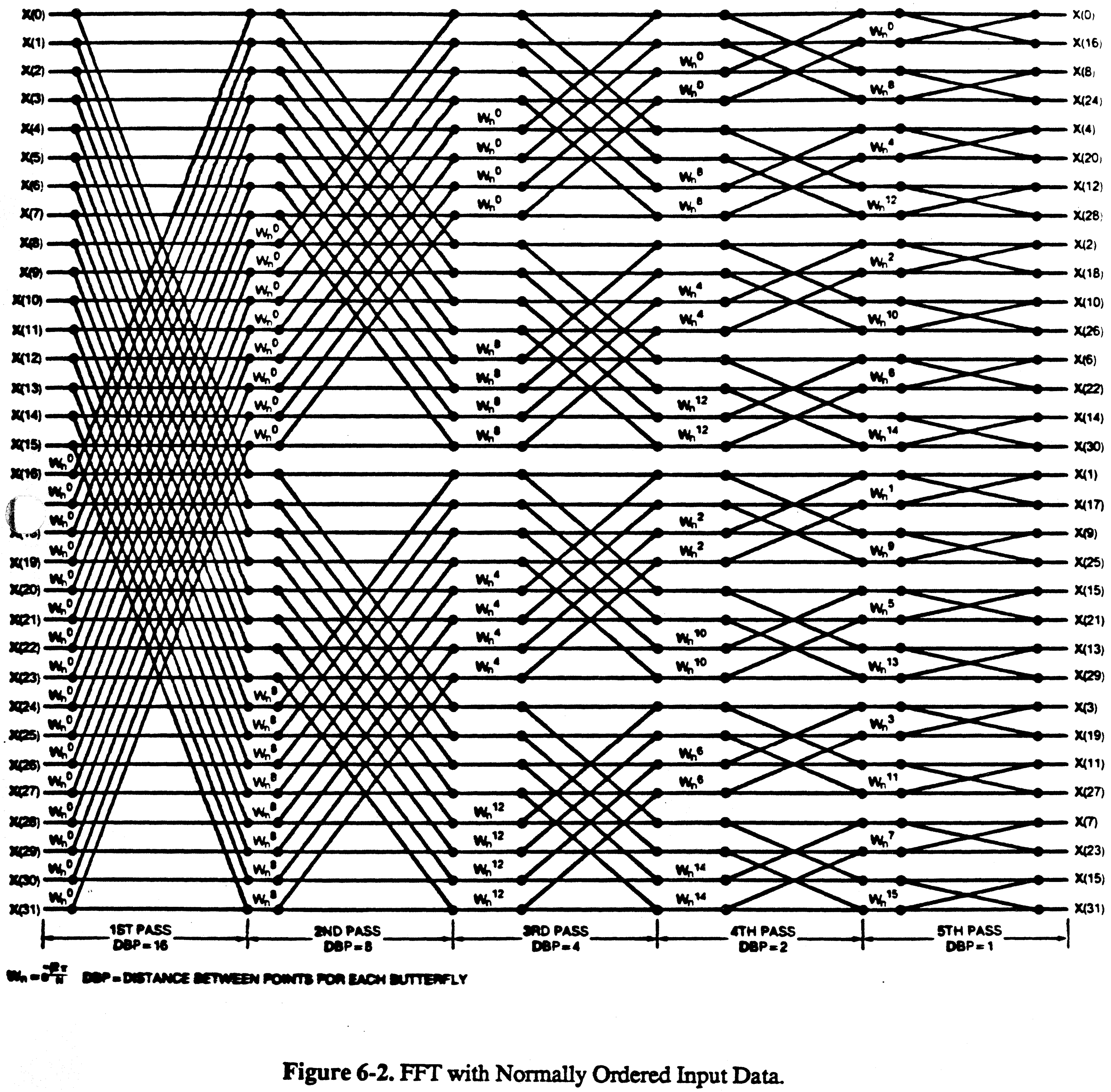

X(0) X(1) X(2) X(3) X(4) X(5) X(6) X(7) X(8) X(9) X(10) X(11) X(12) X(13) X(14) X(15) X(16) X(17) X(18) X(19) X(20) X(21) X(22) X(23) X(24) X(25) X(26) X(27) X(28) X(29) X(30) X(31)
X(0) X(16) X(8) X(24) X(4) X(20) X(12) X(28) X(2) X(18) X(10) X(26) X(6) X(22) X(14) X(30) X(1) X(17) X(9) X(25) X(5) X(21) X(13) X(29) X(3) X(19) X(11) X(27) X(7) X(23) X(15) X(31)
1ST PASS DBP=16
2ND PASS DBP=8
3RD PASS DBP=4
4TH PASS DBP=2
5TH PASS DBP=1
$W_n = e^{\frac{-j2\pi}{N}}$ DBP = DISTANCE BETWEEN POINTS FOR EACH BUTTERFLY

Figure 6-2. FFT with Normally Ordered Input Data.

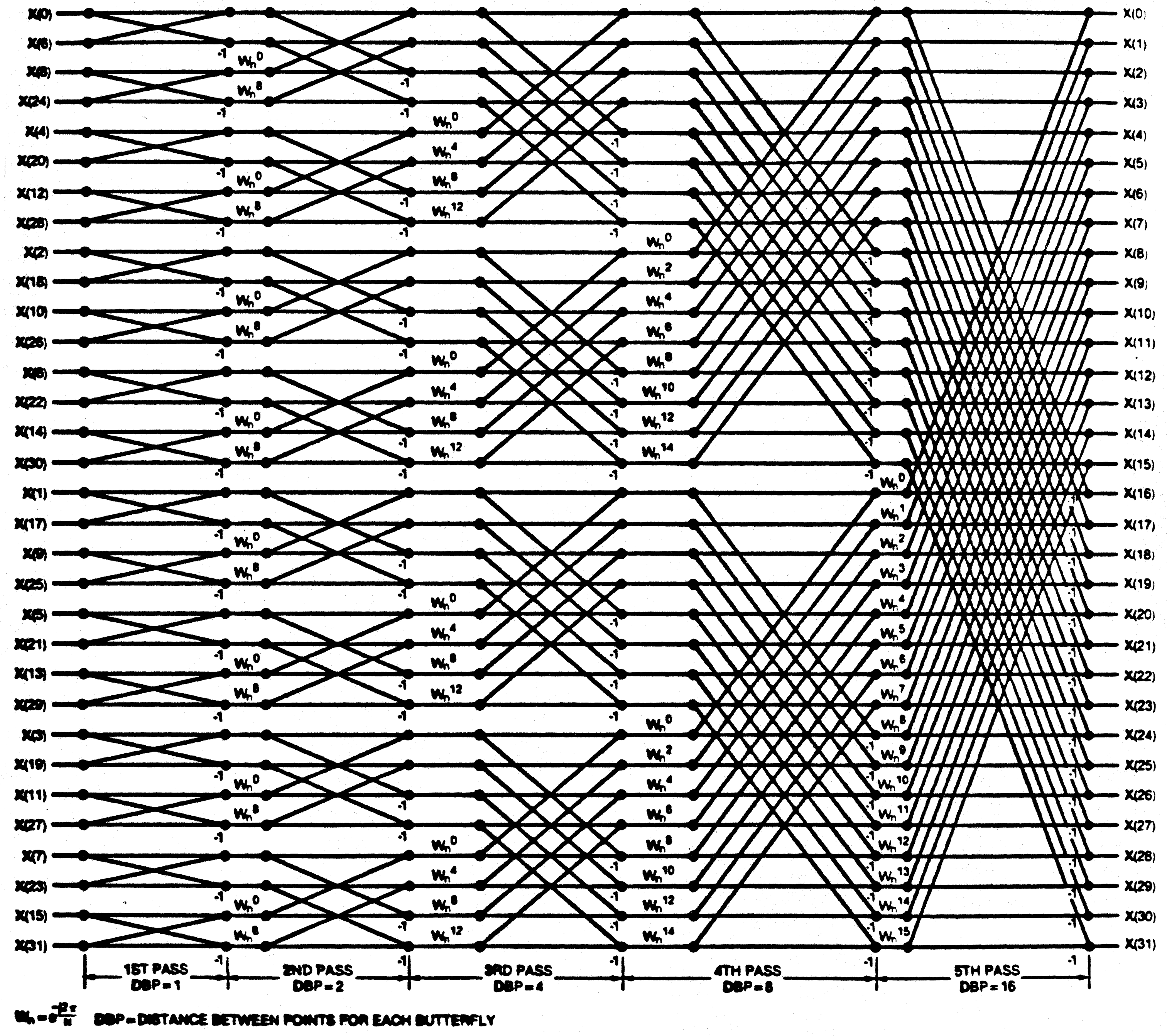


Figure 6-3. FFT with Bit-Reversed Input Data.

6.7.4 Radix-2, Butterflies and Passes

In a radix-2 transform, the number of points in the input sequence must be a power of two. Some of the characteristics of the DIT algorithm determine the number of computations required to compute the FFT. The DIT algorithm of Figure 6-1 is a radix-2 transform as is the VSP FFT instruction. Some of the properties of radix-2 transforms are described below:

1) The number of passes p to complete an N-point radix-2 transform is $p = \log_2 N$, where N is the number of points in the input sequence.

2) The number of butterfly computations b required *per pass* (Figure 6-3) is $b = N/2$.

3) The *total number* of butterfly computations required for an N-point transform is $pb = N/2 \log N$.

The parameter FSIZ in the FFT instruction is the number of points in the transform. NMBT (literal) is the number of butterflies required per pass. Therefore, NMBT (logical) = FSIZ times the number of FFTs of size FSIZ. This applies to both the N-point FFT and the IFFT. This last statement also implies that the VSP is capable of performing multiple FFTs in a single instruction when they are small in size.

6.7.5 Order of Input

An important property of the DIT algorithm is that if the addresses of the input are in normal order, the output addresses are in bit-reversed order, as can be seen in Figure 6-2. The converse is also true, as shown in Figure 6-3, with bit-reversed input and normal-ordered output.

The difference in FFT computation between normal-ordered and bit-reverse ordered input is the distance between the points used to compute the butterfly. In Figure 6-2, note that the first pass inputs to the butterflies are separated by 16 points, while the last pass inputs are separated by one point. In Figure 6-3, the first-pass separation is one point and the last-pass separation is 16 points.

To summarize the above, in an N-point DIT computation:

1) If the input data is in *bit-reverse order*, the output will be in normal order. The distance between first pass inputs to the butterfly is one point; last pass distance is N/2 points.

2) If the input data is in *normal order*, the output is in bit-reverse order. The distance between first pass inputs to the butterfly is N/2 points; last pass distance is one point.

The FPS parameter in the FFT instruction defines the distance between the inputs to the butterflies in the first pass. The LPS parameter defines the distance between the inputs to the butterflies on the last pass. R specifies whether the input is in normal or bit-reversed order. The use of these three parameters allows the flexibility of ordering the input and output data as desired in the application.

FPS, LPS and R must be selected to conform to the two rules stated above. If FSIZ is smaller than NMBT, then FPS and LPS refer to an FFT with FSIZ number of points. This is important to remember when performing multiple FFTs in a single instruction.

The RV parameter in the LD instruction allows data to be read into the VSP in bit-reversed order. The R parameter in the FFT instruction defines this order to the execution unit aboard the VSP. The RV parameter in the ST instruction writes data stored in internal VSP RAM into external memory in bit-reversed order. Use of these three parameters will allow data to be read, computed and stored in the desired normal or bit-reversed format.

6.7.6 Overflows and Block Floating Operation

FFT computations consist of multiplications and additions, where the addition operation may produce overflows. The magnitudes of the numbers in the data sequences generally increase in each pass. The average rate of growth of the magnitude is up to one bit per stage. The FFT instruction offers two scaling techniques to prevent overflows.

One technique is the fixed divide-by-two option which right-shifts the data after each pass. This technique guarantees against overflow *only* if the starting magnitudes of the data samples are each equal to or less than one. This technique is easy to program and fast to execute, but it causes unnecessary loss of accuracy when no overflow occurs. Fixed scaling is implemented in the VSP by setting AS=1 in the FFT instruction.

The second method to prevent overflow during FFT calculations is to use the block floating-point operation that right-shifts the data only if overflow occurs. This technique is more accurate than the fixed divide-by-two operation. The block-floating operation of the FFT instruction can right-shift data by one or two bits in each pass. Block floating-point operation is achieved in the VSP by setting AS=0 in the FFT instruction. This technique should be used for the highest resolution. It also requires the use of the scale commands in conjunction with the FFT instruction.

The use of block-floating operations requires more care when constructing large transforms from smaller ones. Assume that a 256-point FFT is being constructed from four 64-point FFTs, each of which have been computed with different scale factors. In this case, it is required that each of the smaller transforms be normalized to the same scaling factor. The 64-point transform with the maximum scale factor will not need additional scaling. However, each of the other three 64-point transforms must be scaled by a factor equal to the *maximum factor minus the respective scale factor for the 64-point transform.*

The VSP instruction set is tailored for this scaling operation. Using the STI instruction, the scale register containing the four scale factors can be stored into external RAM. The LDSM instruction can load the scale RAM with up to 64 scale factors stored in successive memory locations. Each scale factor takes one four-bit nibble of space; there are four scale factors per word. In the 256-point FFT example, the LDSM instruction would load only one scale word containing the four scale factors, one for each 64-point sub-transform. The SCL instruction then scales the four transforms with the parameter SB=1. This parameter tells the execution unit to scale (right-shift) each vector by the following amount:

> *old maximum scale register minus the appropriate scale RAM value for that 64-point vector.*

These commands will be demonstrated again in more detail with a design example.

6.7.7 The FFT Coefficients

The complex FFT coefficients, designated as W_N^k in Figures 6-2 and 6-3, are essential parameters in FFT computations. Each of the complex coefficients represents an angle O = k/N x 360°. The VSP internal sine-cosine look-up table stores 256 cosine coefficients in one-quarter of a cosine wave, which specify 1024 complex FFT coefficients from 0 to 2PI. The FFT calculation uses the first 512 values from 0 to PI. This is enough for computing up to 1024-point complex transforms without providing external sine-cosine values.

The RBA (ROM Base Address) is used to specify the starting angle used in each pass of the FFT computation. The angle increments used throughout an FFT computation by the VSP are automatically specified through the LPS, FPS, R, FSIZ, NMBT, I and RBA parameters in the instruction.

RBA does not specify a physical address in the ROM. Its logical value (RBA:O x 10) corresponds to the angle O = k/N x 360°, and its literal value RBA = O x 1024/360° corresponds to the offset address of W within the 512 coefficients from 0 to PI.

Figure 6-2 illustrates the use of RBA. The third pass is divided into four eight-point transforms. The first coefficient in each of the 8-point transforms is W_N^0, W_N^8, W_N^4 and W_N^{12}. For N=32, the angles correspond to 0°, 90°, 45° and 135° respectively; the literal RBA values are 0, 256, 128 and 384; the logical RBA values are 0, 900, 450 and 1350. Note that the literal values of RBA in successive passes are obtained by right-shifting the previous literal RBA value. The FFT instruction generates successive RBAs by this right-shift operation.

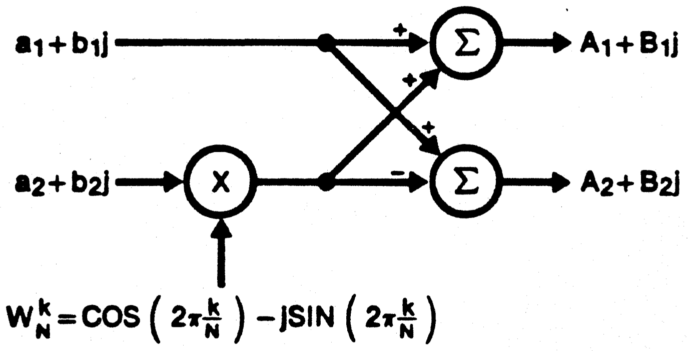

$a_1 + b_1 j$
$a_2 + b_2 j$
X
Σ
Σ
+
+
+
−
$A_1 + B_1 j$
$A_2 + B_2 j$
$W_N^k = \cos\left(2\pi \frac{k}{N}\right) - j\sin\left(2\pi \frac{k}{N}\right)$

Figure 6-4. The FFT Butterfly.

CHAPTER VII

MACRO COMMAND-LANGUAGE

7.1 Overview

The macro command-language provides the user with an alternative to the menu mode of interfacing with the VSPS. When in the command mode (as opposed to the menu mode), the user enters commands by typing them directly at the terminal. To enter the command mode from the menu mode, the user need only choose the command mode option which is available in most of the menus. For instance, the command mode is entered from the main simulator menu by the '10' option. To get back to the menu mode, the user can type 'menu' <cr> which will bring him back to the menu from which he entered the command mode.

All the commands available in the command mode have a uniform syntax and they may have optional arguments. The user can specify the values for the command arguments on the command line; if not specified on the command line, they will assume their default values. A very useful feature of the command language is its *extensibility*. The user can define new commands (macros) as collections of existing commands and macros in order to simplify long and repetitive tasks. These new macros are then added to the macro command library for future use.

Once the macro command mode is entered, the 'HELP' <cr> command will display all of the defined macros. If no new macros have been defined by the user and added to this library, the macros defined by the system are displayed. When macros are defined by the user, they will also be displayed by the 'HELP' command. Descriptions and syntax of all of the functions provided by the macro command language are available by typing 'help*fn*' <cr>, where *fn* is the function name desired. For instance, typing 'help *mulr*' <cr> will describe the *mulr* macro and show the command format.

7.2 Command Syntax

This section describes the syntax of the commands. Each command line requires the command name, followed by optional arguments. Each argument has a defined name, default value and prompt string associated with it. These values are defined when the macro is created. *Numerical arguments* (as opposed to *string arguments*) have minimum and maximum values associated with them. All arguments, when specified on the command line, can be either *named* or *positional*. "Named" arguments are entered by specifying their name and value. Their position on the command line is unimportant. "Positional" arguments are specified by their value only, and their respective position on the command line *is* important. The command interpreter checks each argument that the user enters on the command line. If the argument is not specified, then the command interpreter assigns it the default value. If the argument is specified, the interpreter checks its value and then executes the command.

Command and macro names can be abbreviated to the smallest unique name. For example, "quit" can be abbreviated to "q", provided that no other command or macro starts with a q. Argument names can be abbreviated in the same manner, but the abbreviations must be completely unambiguous with respect to other arguments within the same command.

Default values for non-string arguments can be modified using a double assignment ("==").

An extended Backus-Naur Form (BNF) notation is used to give a formal description of the macro command-language syntax. The BNF syntax is presented in section 7.2.3. The BNF form provides the description of the strict format required within the command-language for creation of macros and execution of commands. The end of this chapter will present a more detailed introduction to the language format as well as an example session.

Prior to defining the formats required for creating new macros, pre-defined macros and commands provided by the system are presented. Studying these commands and macros briefly will make the BNF description more intuitive.

7.2.1 Built-in Commands

The following commands are built into the macro command-language interpreter:

HELP [command] -- Print help information on the terminal. When no argument is given it will print the names of all the commands. When a command name is given as an argument, it will print help information for the specified command.

INCLUDE filename -- Include the specified file in the input stream and interpret its contents as commands. INCLUDEs can be nested. If a MENU command is encountered in the file, then the rest of the file will not be interpreted.

MENU -- Return to the menu mode.

QUIT -- Halt VSPS execution and return to the system.

CLEARMEM [realcomplex, address, n] -- Clear *n* samples of simulated VSP external memory to zero. CLEARMEM is normally used in conjunction with SIGNAL. *Realcomplex* is one if a real signal is to be used, and two if a complex signal is to be used. *Address* is the address of the first point and *n* is the number of points to be cleared.

SIGNAL [address, n, amplitude, phase, f, type, optype] -- Modify the VSP external memory by adding to it, subtracting from it, multiplying it or dividing it by a signal to be generated in the APPENDMEM command. The normal mode of operation is to clear the memory and then add to it as needed. *Address* is the initial address in simulated external memory; *n* is

The normal mode of operation is to clear the memory and then add to it as needed. *Address* is the initial address in simulated external memory; *n* is the number of *points*; *amplitude*, *phase* and *f* are the amplitude, phase and frequency of the signal to be appended. *Type* is the type of signal (1 - sine wave, 2 - square wave, 3 - cosine wave, 4 - random signal, 5 - Flat (DC value), 6 - impulse). *Optype* is the type of operation on the signal (1-add, 2-subtract, 3-multiply, 4-divide).

FFPOP -- Executes user code existing in the vspop function.

MOMENT [n, a, s] -- Calculate the moment of an input vector. *N* is the number of points in the sample, *a* is the starting address in external memory, and *s* is the address of the scratch area. MOMENT calls the *VSP Signal Processing Library* (Menu M-5) and executes option '7'.

MAGCOMPLX [s, i, a, v] -- Calculates the magnitude of a complex vector. *S* is the vector length in samples, *a* is the input base address, *i* is the number of iterations, and *v* is the SH parameter. MAGCOMPLX calls the *VSP Signal Processing Library* (Menu M-5) and executes option '6'.

POWERSPECT [a, s, f] -- Calculate the power spectrum of a vector. *A* is the length of the vector in complex samples, *s* is the base address of the complex input data, and *f* is the base address of the packed real output. POWERSPECT calls the *VSP Signal Processing Library* (Menu M-5) and executes option '4'.

CONVOL [n, sa, a, s, f] -- Calculate the convolution of two one-dimensional sequences. *N* is the number of samples in each sequence, *sa* is the address for the scratch area, *a* is the address of the filter imulse response, *s* is the address of the real data sample, and *f* is the number of real data samples to filter.

PLOT [option, diff, address1, address2, nsamples, memtype, format] -- Plot external or internal memory. Single sequence or a difference of two sequences can be plotted. *Option* is one for complex. two for real part, three for imaginary part and four for packed real. *address1* and *address2* are the addresses of the real and complex parts respectively, *n* is the number of points, and *memtype* is the type of plot (1 - dumb-terminal plot, 2 - print to terminal, 3 - VT240 plot).

VECOPT [realcomplex, address, address2, n, type] -- Perform either a vector addition or vector multiplication operation. *Realcomplex*, *address*, *address2* and *n* are the same as for those in PLOT, above. *Type* is one for addition and two for multiplication. VECOPT calls the *VSP Signal Processing Library* (Menu M-5) and executes option '2' or '3' as a function of the *realcomplex* parameter.

MSAVE [file, address, n] -- Store the signal existing in simulated external memory to a disk file in binary format. The signal can then be loaded using the MLOAD macro. *File* is the file name, *address* is the starting address of the signal in external memory, and *n* is the number of data points to save.

MLOAD [file, address, n] -- Load a signal from a disk file in binary format into simulated external memory. The signal must have been previously saved using MSAVE. All arguments are the same as those described in MSAVE above.

CSAVE [command, file] -- Save a macro as a text file. This macro can later be loaded using the INCLUDE command. *Command* is the name of the macro. *File* is the name of the disk file to which the macro should be saved.

UNDEF [c] -- Undefine a macro or a command. The name of the macro (c) is made available for redefinition.

JEDEC [file, n, address] -- Dump the contents of the simulated external memory into a disk file in JEDEC format. *File* is the disk file name, *n* is the number of points, and *address* is the beginning address in simulated external memory.

XSAVE [file, address, n] -- Store the signal existing in simulated external memory to a disk file in Intel Hex format. The signal can then be loaded using the XLOAD macro. *File* is the file name, *address* is the starting address of the signal in external memory, and *n* is the number of data points to save.

XLOAD [file, address, n] -- Load a signal from a disk file in Intel Hex format into simulated external memory. The signal must have been previously saved using XSAVE. All arguments are the same as those described in XSAVE above.

7.2.2 System Macros

The following are standard system macros. They are loaded automatically upon initialization. They have been created on top of the built-in commands described in section 7.2.1.

All of the system macros defined below have the same definition for the three parameters: *a* and *a2* are the addresses of the real and complex parts respectively, and *n* is the number of points in the vector.

MULR [n, a, a2] -- Multiply two real vectors.

MULC [n, a, a2] -- Multiply two complex vectors.

ADDR [n, a, a2] -- Add two real vectors.

ADDC [n, a, a2] -- Add two complex vectors.

LISTC [n, a, a2] -- Print the values of a complex vector.

PLOTR [n, a, a2] -- Plot a real (imaginary) vector on a "dumb" terminal in character mode.

PLOTC [n, a, a2] -- Plot a complex vector on a "dumb" terminal in character mode.

7.2.3 Backus-Naur Description of the Macro Command-Language

An extended BNF notation is used to give a formal description of the macro command-language syntax. Using this notation, the vertical bar ('|') denotes a choice or logical OR condition, curly braces ('{', '}') denote grouping, and asterisks ('*') denote zero (or more repetitions of the previous construct). Some explanations and semantics are given in 'C' style comments following the syntax rules.

program := { command | definition | emptyline }*
> /* i.e. the input program is a sequence of *commands* and *definitions* */

command := ID optionalarguments <cr>
> /* *ID* is the command name. It has to be followed by *optional arguments* and a carriage return <cr>.*/

optionalarguments := /* empty */ | arguments { ',' argument }*
> /* i.e. the arguments are a list of single *arguments* separated by commas. */

argument := positionalargument | namedargument
> /* *Positional arguments* are associated with their formal parameters through their ordinal position in the command. *Named arguments* are associated by their name.*/

positionalargument := value

namedargument := ID '=' value | ID ':' value | ID '==' value
> /* *ID* is the name of the command argument as specified in the command definition. Three types of assignment are available, logical (=), literal (:) and default modifying. Positional arguments are always assigned logically. */

value := expression | REALV | STRINGV | QMRK | DFLT
> /* The value for an argument can be an integer expression, real constant, or string constant, depending on the type of argument. *Real* numbers must have a decimal point and *strings* must be "quoted". In addition, the value for the argument can be a '?' (QMRK) or a '%' (DFLT). If the value is '?', the user will be asked for the value. If the value is '%', the interpreter will choose the default value as defined in the command definition. */

asked for the value. If the value is '%', the interpreter will choose the default value as defined in the command definition. */

expression := term { '+' term }* | term { '-' term }*

term := factor { '*' factor }* | factor { '/' factor }*

factor := INTV | '(' expression ')'

/* *Expression* is a sum of products. Normal rules of operator precedence are followed, and expressions can be nested. Unary '+' and '-' are not supported, and currently only integers are permitted in an expression. */

definition := startline commandsequence endline

startline := DEFMAC macroname '(' macargs ')' <cr>

endline := ENDMAC { ID } <cr>

macargs := /* empty */ | onemacarg { ',' onemacarg }*

/* The arguments in the definition line are separated by commas. */

onemacarg := argname ':' argdescription ':' prompt

/* *argname* is the name of the argument, and *prompt* is the prompt string written to the terminal if the user desires to be interrogated for the argument by using the '?' symbol.*/

argdescription := 'I' ':' minval ':' maxval ':' defval |
'R' ':' minval ':' maxval ':' defval |
'S' ':' defstring

/* *minval*, *maxval* and *defval* are the minimum, maximum and default values for the arguments. They should be integers for integer arguments (specified by 'I') and real for real arguments (specified by 'R'). */

commandsequence := { macrocommand | helpinfo }*

/* A macro may have any number of commands in its definition. Each command starts on a new line with a '!' (exclamation mark) as its first character. The commands are not executed during definition, only during expansion. */

helpinfo := '##' Rest of the line.

/* This is used to add help information to a macro definition. This information is displayed in response to the HELP command. */

7.3 Macro Examples

In the following examples, the command innterpreter prompts with '>' in the normal macro command mode, and with '>>' when in the macro definition mode. Explanation comments follow the 'C' convention (i.e. /* ... */).

<u>**EXAMPLE 1:**</u>

The first example illustrates how to create a simple macro:

>defmac sameashelp(item:s:thisstring:whatstring)

/* Define a new macro command with name *sameashelp* with one argument of type *string*, default value of *thisstring* and prompt of *whatstring*. */

>>## A simple macro

>>!help $1

/* The first (and only) macro argument is passed to the "HELP" command defined already within the simulator. */

>>endmac sameashelp

/* The macro has only one command, "HELP". */

/* After the macro sameashelp is defined as above, the following commands are equivalent: */

```
>help mulr
>sameashelp mulr
```

Typing either of the above commands will display the help information for the 'mulr' command.

Note that in the macro definition mode all the macro body commands are preceded by '!'. Also in the definition body, all arguments to be passed on the command line are preceded by '$'.

EXAMPLE 2:

This example creates a more sophisticated macro that calls other macros which have been defined by the system. The result from this macro is to create a binary phase-shift keyed signal with a user-specified carrier frequency modulated by a user-specified square-wave frequency. The signal is stored in a disk file with a name provided by the user. After the signal is generated, it is plotted on the terminal.

```
>defmac bpsksig(address:i:0:4095:0:"beginning address",
                        nmpt:i:0:4095:0:"number of points",
                        sqfrq:r:0.0:100.0:0.0:"square wave frequency",
                        sinfrq:r:0.0:100.0:0.0:"sine wave frequency",
                        fname:s:"bpsk.lib":"file name")
>>## Binary phase-shift keyed signal
>>!clearmem 1 $1 $2
>>!signal $1 $2 1.0 0.0 $3 2 1
>>!signal $1 $2 1.0 0.0 $4 1 3
>>!msave $5 $1 $2
>>!plot 1 $1 ? $2 3
>>endmac
```

The first command of the macro defines the name and the five parameters which may be specified. The name of the macro is *bpsksig*. The first parameter, *address*, is the beginning address in simulated external memory where the macro should write the signal generated. The proper response is an integer in the range from 0 to 4095. The default value is 0, meaning that a signal will not be created if a parameter is not specified on the command line. The prompt for the address is *beginning address*. The second parameter, *nmpt*, is also an integer with the same default values as *address*, but with the prompt *number of points*.

The next two parameters, *sqfrq* and *sinfrq*, specify the frequencies of the square wave and sine wave respectively. Both parameters expect real number responses between 0.0 and 100.0 with a default of 0.0 (which represents a DC value). The prompts for the two parameters are *square wave frequency* and *sine wave frequency* respectively. The final parameter, *fname*, represents the disk file name to which the signal created should be written. The default file name is *bpsk.lib*. The prompt for the file name is *filename*.

The first macro call is to CLEARMEM which clears the amount of simulated external memory at the beginning address specified on the command line. The first SIGNAL command generates the square wave and stores it in memory. The second SIGNAL command generates the sinusoid and performs an element-by-element multiply with the square wave just created. The result is the binary phase-shift keyed signal desired. The MSAVE command saves the signal created to a disk file. The PLOT command plots the created signal on the terminal in the high-resolution mode.

The ENDMAC command ends the macro definition. If a 'HELP' command is now executed, the macro we just created will be listed with in addition to all of the macros previously defined by the system. To execute the newly created macro, issue the following command.

```
>'bpsksig' <cr>
```

This will create a BPSK signal with the defaults defined above. In order to change the parameters, issue the same command followed by the respective parameters. For instance,

```
>'bpsksig 0 256 8.0 32.0 bpsk.lib' <cr>
```

will create a BPSK signal with a carrier of 32 modulated by a square wave of frequency eight. 256 samples of the signal will be stored to simulated external memory beginning at address 0. It will also be stored to a disk file called *bpsk.lib.*

7.4 Saving and Re-using Macros

Unless the macro itself is saved in a disk file, it will not be present the next time the simulator is run. Typing the following sequence,

```
>'savecmd bpsksig' <cr>
```

will save the *bpsksig* macro to a disk file with the name *bpsksig*. The next time the simulator is run, the macro can be included in the list of system macros by simply typing:

```
>'include bpsksig' <cr>
```

Issuing a 'HELP' command will now list *bpsksig* with all of the other system macros.

It is also possible to have user-created macros included automatically when the command mode is entered. This is implemented in one of two ways:

1) using the system text editor, create an ASCII file called *usermac*. Within this file, create all of the desired macros to be loaded upon entering the simulator. When the command mode is entered, the interpreter checks for the existence of the *usermac* file and executes the commands resident therein.

2) create user-defined macros interactively from within the command environment as described in the above examples. When they execute properly, issue a 'savecmd *macroname*' <cr> which saves the macro command to a disk file. Then edit (create if it doesn't exist) the *usermac* file, and insert the statement 'include macroname' <cr> in the file. If more than one macro is to be loaded upon entering the command mode, use separate 'include' commands for each macro.

7.5 An Example Session Using the Macro Command Language

Initiating a session using the VSPS by typing 'vsps' <cr> on the terminal brings up the *Main* Menu as shown:

```
VECTOR SIGNAL PROCESSOR SIMULATOR V2.3-5

ZORAN CORPORATION PROPRIETARY SOFTWARE
COPYRIGHT (C) 1986 ZORAN CORPORATION
ALL RIGHTS RESERVED

MAIN MENU

1  HELP
2  VSP instruction tutorial and execution
3  Data generation and display
4  Display options, timing control and queueing
5  Signal processing library for VSP
6  IEEE signal processing library
7  Application library
8  Execute user program in vspop()
9  Execute batch commands and VSPS validation
10 Command mode
11 Exit

Specify value of your selection (0)(1):3
```

- Entering '3' will call the *Data Generation and Display* Menu.
- Entering '8' in the *Data Generation and Display* Menu will call the *Signal Generation* Menu.
- Entering '2' in the *Signal Generation* Menu will query the user with the questions shown below. All queries are responded to with a simple <cr> which selects the default options, except in the second to last question. Because it is desired to write only a single signal to simulated external memory, the user should respond to this query with a '0' to end the signal generation.

```
Specify value of memory base address (-1)(0):
Specify value of total amplitude (negative for real data) ( 1.00000):
Specify value of number of samples (0)(128):
Specify value of 1 for bit-reversed, 2 for all 0 data (-1)(0):
Specify value of 1 - sum of Sines, 2 - special, 3 - saved, (0)(1):
Signal options are:  1- Flat or Step, 2- Impulse, 3- Cosine  4 - Uniform
        Random, 5 - Square Wave, 0 - to Quit
Specify value of signal type (-1)(3):
Specify value of relative amplitude ( 1.00000):
Specify value of phase in degrees (30.00000):
Specify value of cycles per total samples ( 1.00000):
```

```
Signal options are:  1- Flat or Step, 2- Impulse, 3- Cosine  4 - Uniform
        Random, 5 - Square Wave, 0 - to Quit
Specify value of signal type (-1)(3):0
Specify value of 1 thru 10 to save signal (11 through 13 to save to disk) (-
        1)(0):
```

After the signal has been generated, the user is returned to the *Signal Generation* Menu. Typing a <cr> will return operation to the *Data Generation and Display* Menu. A second <cr> will return operation to the main simulator menu. Typing a '10' <cr> from the *Main* Menu will call the command mode.

The '>' prompt at the left margin verifies that the command mode has been entered.

Typing 'help plot' <cr> after the prompt will display the description and information available for the pre-defined plot macro. The response to 'help plot' <cr> is shown below:

```
COMMAND: plot
TYPE: PRIMITIVE
FUNCTION: Plot or list the data in the external memory.
ARGUMENTS:
        realcomplex: INTEGER
        prompt: real(1) complex(2) diff(4) or exit(0)
        min:0, max:4, default:2

        address: INTEGER
        prompt: starting address
        min:0, max:32767, default:0

        address2: INTEGER
        prompt: address2
        min:0, max:32767, default:0

        n: INTEGER
        prompt: number of points
        min:1 max:32768, default:128

        type: INTEGER
        prompt: Type of plot (1 - plot, 2 - print, 3 - graph)
        min:1, max:3, default:1
```

Typing 'plot 1,0,0,20' <cr> will execute the plot macro with the parameters specified. The plot will be displayed on the terminal as shown below:

```
Minimum 0, Maximum 28381, RMS  23172.98: R - real, I - imaginary
0:0....+....|....+....|....+....|....+....|....+....|....+....|....+....|..
0:                                                                   R
1:                                     R
2:                                                                  R
3:                                        R
4:                                                               R
5:                                           R
6:                                                            R
7:                                               R
8:                                                          R
9:                                                  R
10:                                                       R
11:                                                     R
12:                                                    R
13:                                                        R
14:                                                 R
15:                                                          R
16:                                              R
17:                                                             R
18:                                          R
19:                                                               R
```

If there are no additional functions to be performed within the command mode, typing 'menu' <cr> will return operation to the calling menu (the *Main* Menu in this example). Typing '11' <cr> from this point will exit the user from the simulator and return control to the operating system.

CHAPTER VIII

INTRODUCTION TO VSP APPLICATIONS

8.1 Overview

The ZR34161 Vector Signal Processor is designed specifically for high-performance signal processing applications. It is called a *vector* processor because it is most efficient at processing arrays or vectors of data. Examples of vector operations include: performing Fast Fourier Transforms, vector and matrix multiplications or additions, and convolutions and correlations, as well as modulation and demodulation functions. Given an input array or vector of data, the VSP is very efficient at loading the entire array, performing the vector signal processing function on this array, and storing the resulting array back into external memory.

It is the intent of this chapter to provide an introduction to many of the signal processing applications in which the VSP excels. The general ideas behind the applications programmed in the simulator under the *VSP Signal Processing Library* (Menu M-5) and the *Applications Library* (Menu M-7) will also be presented. For additional information and implementation details on these signal processing topics, please refer to additional Zoran VSP publications and application notes.

8.2 The Fast Fourier Transform

The Fast Fourier Transform operation is central to many signal processing applications. In the following FFT sections, high-level explanations will be provided as to how the FFT instruction in the VSP may be used as an efficient "kernel" for building one- and two-dimensional transforms consisting of from two to 64K complex samples. Instruction execution sequences will be shown where appropriate.

8.2.1 Direct Application of Up to 128 Complex Points

One FFT instruction in the VSP can compute radix-2 FFTs of up to 128 complex points in length. The following simple instruction sequence illustrates the simplicity of executing a transform of this size. The program assumes that the input data is a 128-point complex vector (NMPT:128) existing in external VSP memory beginning at address 0 (MBA:0). The entire vector is loaded into internal VSP memory in instruction 1, the FFT of this data is taken in instruction 2, and the signal is stored out to external memory beginning at address 256 in instruction 3. When an FFT is performed on a normally-ordered input signal, the resulting output addresses of the data are in bit-reversed order. The ST instruction writes out the data from the VSP internal memory in bit-reversed order (RV:1) so that it appears in external memory in normal order.

```
1    LD NMPT:128, RV:0, MBA:0
2    FFT NMBT:128, RBA:0, FPS:64, LPS:1
3    ST NMPT:128, RV:1, MBA:256
```

When performing an FFT, log2(N) passes required to complete the transform. This was discussed in greater detail in section 6.7. When the FFT is executed, the FPS and LPS parameters define the number of passes which are to be performed by the instruction. FPS declares that the *first pass spacing* between data samples is 64, while LPS declares that the *last pass spacing* is one sample. There are seven passes required to get from an initial spacing of 64 samples down to a final spacing of one.

Note in the above example that not all parameters have been defined for the three instructions. This was done intentionally so as not to clutter the command lines with too many parameters.

8.2.2 Overlapped Instruction Execution

If continuous transforms of less than or equal to 64 complex points are to be implemented by the VSP, then the internal arithmetic instructions may be overlapped with the I/O instructions to speed the processing. When overlapping instructions, the VSP RAM is split into two 64 complex-word sections; I/O instructions such as LD and ST are performed with one section, while the FFT (or other internal ALU instructions) is performed with the other section. The RS parameter selects the RAM sections on which the instructions operate.

The following nine-instruction sequence executes continuous 64-point FFTs overlapped with memory (I/O) instructions. The concept of this example loop is valid regardless of whether continuous 64-point transforms are being performed, or whether larger transforms are being performed with the 64-point transform as a kernel. Note that as in the last example not all parameters are shown for all of the instructions. In addition, this example does not use the block floating-point capability of the VSP, but merely uses a fixed right-shift after each FFT instruction.

Note that instructions requiring both ALU operations and I/O simultaneously cannot be overlapped with other ALU or I/O instructions. There are four instructions of this kind and they are described in section 6.4.

```
/*      The first two instructions set up the FFT loop. The first instruction loads RAM section 0 with 64 points, while
        instruction 2 begins the FFT on the data just loaded into section 0. Note that no overlapping is performed with
        these two instructions.

*/

                1       LD NMPT:64, RS:0
                2       FFT NMBT:64, RS:0

/*      Now load the second section of RAM while the first section is being used by the ALU for executing the FFT.
        Note that the control section internal to the VSP knows to begin executing instruction 3 just as soon as
        instruction 2 has begun execution. Instruction 3 is the beginning of the FFT loop.
*/
        LOOP:   3       LD NMPT:64, RS:1

/*      The next FFT instruction (4) does not begin execution until the previous FFT instruction (2) completes
        execution. This is because the LD instruction (3) takes much less time to execute than the FFT instruction (2), so
        the VSP waits until the first FFT is finished. After the second FFT (4) begins on the data just loaded into RAM
        section 1, the VSP control unit immediately begins execution of the ST (5) instruction which writes out the
        previously transformed data from RAM section 0.
*/
                4       FFT NMBT:64, RS:1
                5       ST NMPT:64, RS:0

/*      The ST instruction (5) completes execution before the FFT instruction (4), so the control unit in the VSP
        immediately begins execution of the LD instruction (6) which loads the data into RAM section 1 in preparation
        for the FFT in instruction 7. FFT (7) does not begin execution until FFT (4) is complete.
*/
                6       LD NMPT:64, RS:0
                7       FFT NMBT:64, RS:0

/*      Store the results from the FFT (4) while FFT (7) is in progress.
*/
                8       ST NMPT:64, RS:1

/*      The basic FFT loop is now complete. JMPI commands an indirect jump back to instruction 3 to begin the next
        sequence of transforms. Note that FFTLOOP is an address which points to LOOP, and its value is not described
        in this example.
*/
                9       JMPI FFTLOOP
```

8.2.3 Groups of Small Transforms

The high functionality of the FFT instruction allows the calculation of multiple transforms in parallel with one FFT instruction. The only restrictions are that all of the small transforms must be the same size, and that the total number of points in all the transforms be less than or equal to 128 complex points. Up to eight 16-point transforms may be calculated in parallel using a single 128-complex-point FFT instruction. The FSIZ parameter in the FFT instruction is specified to equal the size of each of the small transforms. In this case, FSIZ = 16. The FPS and LPS parameters are specified for a 16-point transform, not for a 128-point transform. The following sequence executes eight 16-point complex FFTs in parallel:

```
1       LD NMPT:128, MBS:16, MSS:16, RV:0
2       FFT NMBT:128, FSIZ:16, RBA:0, FPS:8, LPS:1
3       ST NMPT:128, MBS:16, MSS:16, RV:3
```

When small FFTs are performed in parallel, all transforms performed in the same pass are scaled by the same scaling factor (if the SCL instruction is being used). Because of this, the accuracy on each individual 16-point transform may be less than if that same 16-point transform were calculated by itself. Note that the ST instruction in the example performs bit-reversal within each 16-point block, so all eight transforms are stored out in memory in normal order.

8.2.4 FFTs Larger Than 128 Complex Points

FFTs larger than 128 complex points can be easily computed by the VSP using the FFT instruction as a building block. The larger FFTs must be decomposed or factored into smaller FFT blocks of 128 points or less. As a demonstration, a 256-point complex FFT is shown using four 64-point complex FFTs followed by 64 four-point complex FFTs. Only eight calls to the FFT instruction are needed. Because 64-point transforms are being used, the FFT calculations may be overlapped as described in the continuous 64-point transform example in section 8.2.2. The mathematical derivation and programming details of this process are described in Zoran's *Vector Signal Processor FFT Handbook.*

A way to visualize the decomposition of the 256-point transform is to observe the following grid:

```
x0  x4  x8  x12. . . . . . . . . . . . . . . . .x252

x1  x5  x9  x13. . . . . . . . . . . . . . . . .x253

x2  x6  x10  x14. . . . . . . . . . . . . . . . .x254

x3  x7  x11  x15. . . . . . . . . . . . . . . . .x255
```

Notice that we have, in effect, created a two-dimensional representation of the one-dimensional 256-point transform; each of the four rows contain 128 samples. This procedure involves calculating four 64-point row transforms, followed by 64 four-point column transforms to complete FFT. All transforms larger than 128 complex points may be factored in this way. The program for this transform is shown below. Instruction defaults have been defined prior to the VSP instructions.

```
                DEFAULT MDF:3, AD:0, EI:0, ZP:0, INTRP:0, ZR:0, AS:1, I:0;
/*      This section of code calculates the four 64-point row transforms and stores the results back out to external
        memory.

                LD      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:0,  MBA:IN;
                FFT     NMBT:64,  RS:0,  FSIZ:64, FPS:32, LPS:1, RBA:0;
                LD      NMPT:64,  RS:1,  MBS:1,  MSS:4,  RV:0,  MBA:IN+2;
                FFT     NMBT:64,  RS:1,  FSIZ:64, FPS:32, LPS:1, RBA:0;
                ST      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:0,  MBA:IN;
                LD      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:0,  MBA:IN+4;
                FFT     NMBT:64,  RS:0,  FSIZ:64, FPS:32, LPS:1, RBA:0;
                ST      NMPT:64,  RS:1,  MBS:1,  MSS:4,  RV:0,  MBA:IN+2;
                LD      NMPT:64,  RS:1,  MBS:1,  MSS:4,  RV:0,  MBA:IN+6;
                FFT     NMBT:64,  RS:1,  FSIZ:64, FPS:32, LPS:1, RBA:0;
                ST      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:0,  MBA:IN+4;
                ST      NMPT:64,  RS:1,  MBS:1,  MSS:4,  RV:0,  MBA:IN+6;

/*      This section of code calculates the 64 four-point column transforms and stores the results back out to external
        memory; results are bit-reversed within each block.
*/
                LD      NMPT:64,  RS:0,  MBS:64, MSS:64, RV:0,  MBA:IN;
                FFT     NMBT:64,  RS:0,  FSIZ:64, FPS:2,  LPS:1, RBA:0;
                LD      NMPT:64,  RS:1,  MBS:64, MSS:64, RV:0,  MBA:IN+128;
                FFT               RS:1,                          RBA:16;
                ST      NMPT:64,  RS:0,  MBS:64, MSS:64, RV:0,  MBA:SCRATCH;
                LD                RS:0,                          MBA:IN+256;
                FFT               RS:0,                          RBA:8;
                ST      NMPT:64,  RS:1,  MBS:64, MSS:64, RV:0,  MBA:SCRATCH+128;
                LD                RS:1,                          MBA:IN+384;
                FFT               RS:1,                          RBA:24;
                ST      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:1,  MBA:IN+2;
                ST      NMPT:64,  RS:1,  MBS:1,  MSS:4,  RV:1,  MBA:IN+6;
                LD      NMPT:64,  RS:0,  MBS:64, MSS:64, RV:0,  MBA:SCRATCH;
                ST      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:1,  MBA:IN;
                LD      NMPT:64,  RS:0,  MBS:64, MSS:64, RV:0,  MBA:SCRATCH+128;
                ST      NMPT:64,  RS:0,  MBS:1,  MSS:4,  RV:1,  MBA:IN+4;
```

The RBA parameter assumes values other than 0 in the second set of FFT instructions because the FFT instructions are completing the final four passes required of the 256-point FFT. The RBA parameter is determined by observing a flowchart for the particular FFT being performed. The VSP ROM contains enough coefficients for up to 1024-point transforms; for larger transforms, the coefficients will need to be provided by an external look-up table.

Note that the 256-point FFT can also be computed by two 128-point row FFTs followed by 128 two-point column FFTs, or similarly by eight 32-point row FFTs followed by 32 eight-point column FFTs, or any combination of row and column transforms that equal 256 points. Other large transforms may also be decomposed in a similar manner. The signal processing option in the simulator (Menu M-5) which computes general FFTs greater than 128 points are computed in the same way.

The demonstration 256-point FFT in the *VSP Signal Processing Library* is an example of a two by 128 decomposition of the 256-point FFT.

To summarize, the general procedure for constructing transforms greater than 128 points is outlined below:

1. Decompose the data into columns and rows as above.
2. Perform row transforms.
3. If block floating operation is used, scale all row transforms to one common scale factor.
4. Figure the ROM coefficients for the passes needed to complete the column transforms. Specify the proper RBA and multiply by the proper FFT coefficients where necessary.
5. Do the column transforms.
6. If block floating operation is used, scale all column transforms to one common scale factor.
7. Store the output of the transforms in normal order.

8.3 Additional Discussions About the VSP Signal Processing Library

The following sections contain additional discussions of some of the signal processing algorithms which are contained in the *VSP Signal Processing Library* (Menu M-5) as well as the *Applications Library* (Menu M-7).

8.3.1 Real FFTs

The term "real FFT" means the computation of a complex FFT from a purely real sequence containing no imaginary components. The input is a real sequence, but the output is a complex FFT. There are advantages in certain applications to performing calculations involving real FFTs. Real FFTs can be computed from the complex FFT instruction in the signal processing library in two ways: 1) two N-point real FFTs can be calculated simultaneously using one N-point complex FFT, or 2) one 2N-point real FFT can be calculated with one N-point complex FFT.

Both of the applications discussed in this section are included in the *VSP Signal Processing Library* (Menu M-5), as well as in the *IEEE Signal Processing Library* (Menu M-6).

The first method of two N-point real FFTs from one N-point complex FFT:

1. Functions h(n) and g(n) are real; n = 0,1, ... ,N-1

2. Form the function y(n) = h(n) + *j*g(n); n = 0,1, ... ,N-1

3. Compute the FFT of y(n):

   Y(k) = y(n)*exp[-*j*2*pi*k*n/N] = R(k) + *j*I(k),

   where R(k) and I(k) are the real and imaginary parts of the transform Y(k).

4. Compute the transform of h(n) and g(n) from R(k) and I(k):

   H(k) = [R(k)/2 + R(N-k)/2] + *j*[I(k)/2 + I(N-k)/2]

   G(k) = [I(k)/2 + I(N-k)/2] - *j*[R(k)/2 - R(N-k)/2];

   k = 0,1, ... ,N-1,

   where H(k) and g(k) are the real transforms of h(n) and g(n).

The second method of one 2N-point FFT from one N-point complex FFT:

1. Function x(n) is real; n = 0,1, ... ,2N-1

2. Divide x(n) into two functions:

$h(n) = x(2n)$ and $g(n) = x(2n + 1)$; $n = 0,1, \ldots ,N-1$

3. Form the function $y(n) = h(n) + jg(n)$ $n = 0,1, \ldots ,N-1$

4. Compute the transform of y(n):

   $Y(k) = y(n)*\exp[-j2*pi*k*n/N] = R(k) + jI(k)$;

   $k = 0,1, \ldots ,N-1$,

   where R(k) and I(k) are the real and imaginary parts of the transform Y(k).

5. Compute:

   $Xr(k) = [R(k)/2 + R(N-k)/2] + \cos[I(k)/2 + I(N-k)/2]$
   $-\sin[R(k)/2 - R(N-k)/2]$; $k = 0,1, \ldots ,N-1$

   $Xi(k) = [I(k)/2 - I(N-k)/2] - \sin[I(k)/2 + I(N-k)/2]$
   $-\cos[R(k)/2 - R(N-k)/2]$; $k = 0,1, \ldots ,N-1$

   where Xr(k) and Xi(k) are the real and imaginary parts of the FFT of the 2N real points of x(n).

8.3.2 Fast Convolution and Correlation

This application performs fast convolution or correlation using the FFT. It is based on the fact that convolution in the time-domain is equivalent to multiplication in the frequency domain. The algorithm is designed to convolve or correlate a small real vector of length N samples with a second real vector as large as desired (but with a number of data points equal to some multiple of N). The length of the output will be the difference bewteen the lengths of the two input vectors plus '1'. Notice that circular convolution is not being performed with this technique; this is why the resulting convolution output is not equal to the sum of the lengths of the two vectors minus '1'. The limits on N are 16 to 32 points.

The fast convolution application discussed in this section is included in the *VSP Signal Processing Library* (Menu M-5).

The basic algorithm is as follows:

1 The N-point impulse response of the filter is copied to the real part of a complex vector; the imaginary part remains zero. If correlation rather than convolution is desired, the complex vector is copied in the reverse order.

2 N zeros are appended to this vector making a new impulse response vector of length 2N points.

3 A complex FFT is performed on the 2N-point impulse response vector. The result is saved.

4 The larger vector is broken into 2N-point vectors that overlap by N points. 2N points of the larger vector are copied to the real part of a complex vector. The next 2N points are copied to the imaginary part of the complex vector starting at the Nth point. This loading of overlapped vectors into a complex vector allows two N-point real FFTs to be computed in parallel.

5 A 2N-point FFT of the complex vector is computed.

6 A complex multiplication is performed with the results of steps 3 and 5.

7 An inverse FFT of the product in step 6 is performed.

8 The complex vector result of step 7 contains the first N output points of the convolution in the last N points of the real part, and the next N output points of the convolution in the last N points of the imaginary part.

8.4 Additional Discussions About the VSP Applications Library

The VSP Applications Library currently contains two complete applications: a 16K-point FFT and a Doppler-shift application. The following sections will introduce the concepts of these two applications.

NOTE: These two applications are not currently available under the PC release (version 2.3-5) of the VSP simulator.

8.4.1 A 16K-Point FFT Application

This 16K-point FFT application is included in the VSP Applications Library as Menu M-7-2. Multiple VSPs (one, two, four or eight) can be used in the application to improve computation time. At critical points during the execution of the FFT, users are queried with the option of displaying any part of VSP memory using the plotting facilities of the VSPS.

After the FFT is complete, any number of the resulting transform points may be displayed in order of magnitude. The ordering begins at the point with the largest magnitude. After this, any additional part of the resulting memory can be displayed. The results can be written to a disk file for later reference. Because the vector size in the VSP is limited to 128 complex samples, the FFT must be broken into smaller FFTs and complex multiplications as described in the 256-point FFT example in section 8.2.4. Complex multiplications are required in this application because the VSP contains a sine/cosine look-up table equivalent to 1024 samples, and the FFT size is 16K points.

Following the ideas outlined in Section 8.2.4, the 16K-point transform can be visualized as carrying out the complete FFT by viewing the 16384 points as an array of 128 by 128 points (as shown in Figure 8-1, below). In this figure, the numbers in the grid are the indices of the complex data points. First 128 128-point FFTs are performed on the columns of the array. Then these results are multiplied by the twiddle factors. Next, 128 128-point FFTs are carried out across the rows of the array.

```
------------------------------------------------------
|   0  |   1  |   2  |...                ...  |  127  |
------------------------------------------------------
|  128  |  129  |  130  |...              ...  |  255  |
------------------------------------------------------
|                                                    |
|                          .                         |
|                          .                         |
|                          .                         |
|                                                    |
------------------------------------------------------
|16256  | 16257  | 16258  |...            ... | 16383  |
------------------------------------------------------
```

Figure 8-1. Data Array For A 16K-Point FFT.

This program assumes the following configuration for the architecture of the VSP: one, two, four or eight VSPs in parallel on a bus with 64K RAM and a host processor. The signal generator of the simulator is called first in order to interactively create any test signal desired.

The first set of FFTs is done down the columns of the array; that is, each FFT loads 128 points, with the starting addresses being 0, 1, 2, 3, ... 127, for each column vector. Using the VSP LD and ST instructions, the loading and storing of data is performed with MBS:1 and MSS:128 for each instruction while MBA=0, 1, 2, ... 127. Note the bit-reversal being performed during the store instruction. This restores proper ordering to the results of the decimation-in-time FFT.

The VSP uses block floating-point arithmetic during the FFT computation. This means that the result of each FFT instruction loads the VSP scale register with the number of right-shifts (divide-by-twos) required during that particular transform to avoid arithmetic overflow. This number is entirely dependent on the input data. The final transform results of the columns must have a common scale factor before any processing across the rows can be performed. The next step does this scaling and multiplies the intermediate result by the twiddle factors. The scaling is done by reading the maximum scale factor register and subtracting from it the scale value found for each column transform. Then each column is shifted by this scale difference while being multiplied by the twiddle factor.

The second set of FFTs is then done across the rows of the array. In this case, each FFT loads 128 consecutive points, starting at addresses 0, 128, 256, ... 16256. Using the VSP LD and ST instructions, the loading and storing of data is done with MBS:128 MSS:128, while MBA=0, 128, 256, ... 16256. Again, bit-reversal is used to correct the ordering of the results.

Another pass is then made to get the scale factors across each row to be the same, just as was done for the columns.

Finally, the results are transposed using a series of loads and stores. The transpose maps memory location 128*11 + 12 into memory location 128*12 + 11 and vice-versa.

8.4.2 Doppler Shift Application

Menu M-7-1 includes a Doppler-shift application corresponding to the block diagram contained in Figure 8-2. The purpose of the application is to calculate the frequency shift (or drift) of a carrier of known frequency. The source of the carrier and the cause of its frequency drift are independent of the application. As an example, a satellite ground-station receiving a signal from a satellite circling the earth in a non-geosynchronous orbit will need to track the frequency shift of the carrier. This application is designed to determine the frequency shift. Note that the numbers used for the example in this text are arbitrary, and any other numbers may be used depending on the particular application.

This application first demodulates a received signal down to baseband with the known (unshifted) center frequency of the transmitted signal. After demodulation, the sampling-rate of the signal can normally be decimated (reduced) by a user-specified factor. As an example, assume that the carrier of the satellite is 1MHz, the frequency drift about the carrier is 30kHz, and the carrier is digitized at 4MHz. After demodulation, the 4MHz sampling of the 30kHz baseband signal is excessive; its sampling rate can be decimated. In this example, assume that decimation is performed by a factor of 64; this reduces the 4MHz sampling rate down to 62.5kHz. The decimated signal is then placed in a buffer with a length of 128 complex points. The power spectrum of the complex input vector is taken by performing an FFT and then a magnitude square operation. Finally, the peak component of the power spectrum is found and its bin number is printed out. The system is shown in Figure 8-2 below.

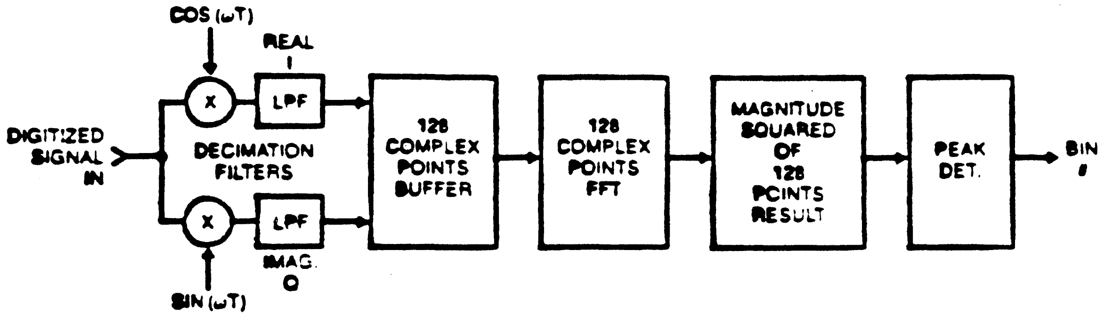


Figure 8-2. Doppler-Shift Application Block Diagram.

The complex demodulation, filtering and decimation are all implemented using a single MLTR instruction. In this example, assume that the low-pass filter is 64 taps in length, and that decimation is also by a factor of 64. In an initialization routine as the system is powered on, the low-pass filter coefficients are multiplied component-wise with the sinusoidal demodulation coefficients and saved in external memory. A single MLTR instruction then corresponds to both demodulating and low-pass filtering of the input signal. The decimation result is taken as the output of the real and imaginary accumulators after the MLTR, corresponding to the summation of the demodulated and filtered signal. Thus, for every 64 samples read into the VSP for demodulation and filtering, a single output sample is generated corresponding to a decimation by 64.

The results of the vector multiplies are placed in a 128-point external buffer. After 128 MLTR instructions, the buffer is full and the FFT and subsequently the MGSQ of the buffer is taken. This corresponds to the power spectrum of the baseband signal, which after decimation was sampled at 62.5kHz. After the magnitude square of the signal is calculated, the peak in the spectrum corresponds to the Doppler-shift of the carrier.

CHAPTER IX

CREATING USER PROGRAMS AND LINKING TO THE VSP SIMULATOR

9.1 Overview

The VSP simulator provides a powerful engineering environment designed to allow complete system-level applications development. This includes algorithm development and verification, test signal generation, modeling of VSP arithmetic, generation of JEDEC files for burning directly into PROMS, the ability to model the world external to the VSP and high-level software modeling of a host processor which may control VSP activity. Many of these functions have already been discussed in this manual.

Up to this point in the manual, all of the interactive features of the simulator have been discussed in detail. The process of writing and ultimately executing user-developed programs has not been covered. This section will discuss the process of writing, parsing, compiling and linking user-generated programs integrating both high-level language simulations of the external environment and VSP instructions into the same simulation.

The following steps highlight the process of writing, parsing, compiling and linking programs with the VSP simulator. Figure 9-1 shows figuratively the same process.

1. Algorithms are sometimes (at least partially) tested in the interactive (tutorial) mode of the simulator (option '2' from the main simulator menu).

2. Signal processing algorithms are written using the VSP instruction set and combined with algorithms simulating the host or external environment written in 'C' or a 'C' callable language.

3. The VSPS parser is used as a preprocessor to translate the combined 'C'/VSP language program into a compilable 'C' program.

4. The system 'C' compiler is used to create an object program from the VSPS parser output.

5. The linker is used to tie together the separately compiled modules, to coordinate cross-references, and to create an executable load module.

Languages other than 'C' (such as Fortran or Pascal) can be used for external environment simulation as long as they can be called from a 'C' main program.

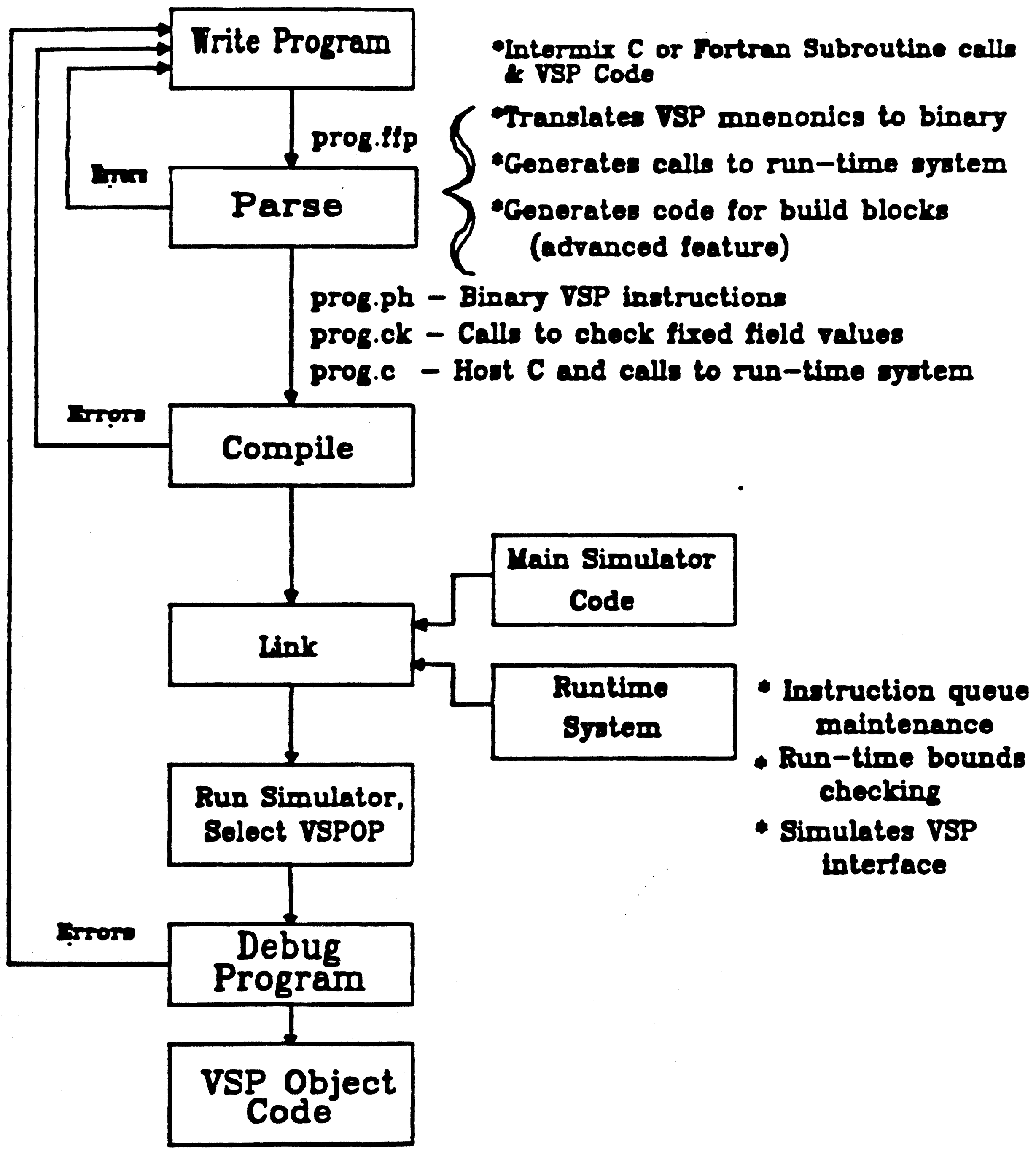

Write Program
*Intermix C or Fortran Subroutine calls & VSP Code
prog.ffp
Errors
Parse
*Translates VSP mnenonics to binary
*Generates calls to run-time system
*Generates code for build blocks (advanced feature)
prog.ph - Binary VSP instructions
prog.ck - Calls to check fixed field values
prog.c - Host C and calls to run-time system
Errors
Compile
Main Simulator Code
Link
Runtime System
* Instruction queue maintenance
* Run-time bounds checking
* Simulates VSP interface
Run Simulator, Select VSPOP
Errors
Debug Program
VSP Object Code

Figure 9-1. The VSP Simulator Program Development Cycle.

9.2 Simple Example

The following program is a relatively simple example designed to illustrate the philosophy of system and algorithm development possible using the VSP simulator. It uses a minimum of VSP and 'C' programming features, yet it gives an indication of how VSP instructions can be used within a high-level 'C' program. It will be referenced in later sections of this chapter to illustrate the parsing, compiling and linking required to create a complete applications program.

This program creates a complex 64-sample single-cycle square wave test signal (in 'C') and saves the signal in simulated external VSP memory indices 0 to 63. Then the FFT of the signal is taken by the VSP. The real and imaginary parts of the FFT are displayed on the terminal (from 'C'). Then the square of the magnitude (power spectrum) is calculated by the VSP and displayed by a 'C' program. Comments are written into the program in traditional 'C' style by enclosing them between the '/*' and '*/' delimiters.

```
/*                        program = example.ffp              */

#include "ffp.h"
#define EXTMEM zrmexter->exmem
/#D#/

vspop()                              /*     Main simulator routine - must be named vspop()       */
{
          int i;                     /*     Only variable declared within vspop()                */

/*   Define the default parameters to be assigned for all instructions. If a parameter is not specified on the instruction line,
     the default values will be used. If a parameter is specified both on the command line and in the defaults, the parameter
     on the command line takes highest priority.
*/

/#
          DEFAULT NMPT:64, RS:0, MDF:3, INTRP:0, ZP:0, MBS:128, MSS:128;
          DEFAULT ZR:0, AD:0, EI:0, RV:0, MGSQ.ADF:2;
          DEFAULT FFT.RBA=0, AS:0, R:0, I:0;
#/

/#E#/

/*   Generate one cycle of a 64 complex-point square wave signal in 'C' and store it in simulated external memory. The
     even indices are the real part of the signal, and the odd indices are the imaginary part.
*/

          for (i=0;i<64;i++)
            {
              if (i<32) EXTMEM[2*i] = 16384;
              else EXTMEM[2*i] = -16384;
              EXTMEM[2*i+1] = 0;
            }

/*   Using VSP instructions, load the entire 64-point complex vector into internal VSP memory, take the FFT, and store the
     results back out to external memory. The STI instruction stores the scale value calculated during the FFT out to address
     512.
*/

/#
          LD MBA=0;
          FFT NMBT:64, FSIZ:64, FPS:32, LPS:1;
          ST RV:1, MBA=128;
          STI NMPT:1, OR=1, STR:5, MBA=512;
#/
/*   Using 'C', display the results of the FFT instruction stored in simulated external memory. Also display the scale
     register result stored in external memory.
*/

          for (i=0;i<64;i++)
                              printf("real(%d)=%d, imag(%d)=%d\n", i, EXTMEM[2*i+128], i, EXTMEM[2*i+129]);
          printf("scale register= %x\n",EXTMEM[512]);

/*   Calculate the square of the magnitude of the FFT (power spectrum). The FFT results still exist inside the VSP, so they
     do not have to be read in from external memory. After the MGSQ, write the power spectrum results out to external
     memory beginning at physical address 256.
*/

/#
          MGSQ;
          ST RV:1, MBA=256, MDF=2;
#/
/*   Using 'C', print out the results of the power spectrum.
*/

          for (i=0;i<64;i++)
             printf("magnitude of bin%d=%d\n", i, EXTMEM[i+256]);
}
#
```

9.3 Mixed 'C'/VSP Language Programming Constructs

This section uses the example in section 9.2 to describe the constructs used in writing mixed 'C' and VSP language programs. It is not the purpose of this section to teach the 'C' programming language to inexperienced users; hence, this section will concentrate only on those constructs needed to write the combined 'C' and VSP code for the VSPS parser. In fact, if languages other than 'C' are going to be used for the majority of the simulation, it is not necessary to fully learn the 'C' language. It is only necessary to use the constructs contained in the example as a template for creating programs. A brief introduction to the 'C' programming language is given in Appendix C.

9.3.1 #include

The statement:

```
#include "/da/ffp/release/ffp.h"
```

loads a set of 'C' declarations, such as the declaration of arrays containing VSP external memory. This statement is required, and its particular form will depend on what system the VSP simulator is running and how the VSPS was installed in the directory structure of the host computer. When running on VAX computers, either "ffp.h" must exist in the current directory, or the directory path must be specified in the include statement as shown in the example. If questions exist about the path where "ffp.h" exists, they should be directed to the system manager. In the example of section 9.3, the directory path of "ffp.h" is */da/ffp/release/*.

PC versions of the simulator have similar requirements for declaring the path of "ffp.h". However, a "set include" command may be issued as described in the PC installation procedure which may eliminate having to declare the path of "ffp.h" within the program if it does not exist in the current directory.

The general form of using the "#include" command is:

#include "*/install_path/*ffp.h"

where *install_path* is the path where "ffp.h" is located.

9.3.2 #define

The statement:

```
#define EXTMEM zrmexter->exmem
```

makes the data structure defining external memory in the simulator easier to use. It allows external memory to be addressed as an array such as 'EXTMEM[address]'. This form was used in the example in section 9.2. 'EXTMEM' is an arbitrary variable name chosen for this example; any valid 'C' variable name may be used in place of 'EXTMEM'.

9.3.3 Delimiters

The end of external declarations must be shown with the '/#D#/' marker. Similarly, the '/#E#/' marker defines to the VSPS parser where executable code begins. The end-of-program marker '##' must be included at the very end of the VSP code module.

Comments in 'C' are delimited by the markers '/*' and '*/'. The example has made extensive use of the comment delimeters outside of the blocks of VSP instructions. It is possible to embed comments within blocks of VSP code as well by using the same comment delimeters.

Source programs as in the previous example may contain both high-level 'C' code as well as VSP instructions. It is up to the VSPS parser to distinguish between the two types of code which may be present. The distinction is made by the use of the two delimiters '/#' and '#/', indicating the start and finish respectively of VSP instructions. The VSPS parser passes unchanged all code outside these markers. Code within the markers must be valid VSP instructions or comments. The VSP instructions are translated into external 'C' function calls to the instruction library by the VSPS parser.

Note in the example that within the first and last program delimiters ('{' and '}'), VSP code exists in two different places, and 'C' code exists in three different places. This can be repeated as many times as desired by simply using the VSP instruction delimeters as previously defined.

9.3.4 The vspop() Function

Vspop() is the required name of the main user function in the combined 'C'/VSP program. The VSP simulator calls the vspop() function to execute the user program when option '8' from the *Main* Menu is executed. The vspop() subroutine may in turn invoke other 'C' subroutines using standard 'C' function calls. Vspop() can also be used to call a subroutine in a language other than 'C', where that subroutine may invoke other subroutines containing VSP code embedded in 'C'.

The left brace '{' is a required 'C' construct defining the beginning of the vspop() subroutine. The last right brace '}' in the program defines the end of the vspop() subroutine.

Note that all variables in a 'C' program must be declared prior to their use. All such variables may be used freely in both 'C' and VSP code. In the example, the *int i* command defines an integer with the name *i*.

9.3.5 Specifying Instruction Parameters

The parameters for each VSP instruction can be set in a number of ways. The previous example illustrates two of those ways. The first way to declare parameters is to use the DEFAULT construct as is shown in the example. Towards the beginning of the program (but inside the vspop() function), the DEFAULT command defines all parameter defaults. The second way to declare parameters is to specify them on the VSP instruction command line. If parameters are not specified on the instruction line, the parser looks through the DEFAULT parameters to see if they have been defined. If parameters are specified on the command line, they will override the default values. As an example of this, if the ST instruction executed after the MGSQ instruction defines MDF=2, and in the default parameters MDF was defined as 3, then the assignment of MDF=2 takes precedence because it exists on the command line. The third way of specifying instruction parameters is to define them using any valid 'C' arithmetic expression. For applications using multiple VSPs, the simulator allows specifying which VSP the instruction is directed to.

Parameters may be defined as either literal or logical values by using the colon (:) for logical values and the equal sign (=) for literal values. Descriptions of the differences between logical and literal values are included in section 4.3.1. The example in section 9.2 uses both logical and literal parameter specifications.

9.4 The VSPS Parser

The VSPS parser is a program that translates a source file made up of both VSP instructions and high-level 'C' code into a form suitable for compilation by the system 'C' compiler. High-level 'C' programming statements outside the'/#' and '#/' delimeters are not affected by the VSPS parser. The parser identifies VSP instructions embedded within the high-level program and translates these instructions into external function calls to the VSP instruction library. The output of the parser is a standard 'C' program containing the original 'C' source code and the external function calls. This parsed output is then run through the system 'C' compiler to create an object module. The object module is then linked with the instruction library which creates an executable module ready to run.

The use and syntax of the VSPS parser must be understood in order to use the VSP simulator to its fullest potential. Parser options are discussed in more detail later in this chapter. Formal constructs are discussed in Appendix D. This section will show how to issue the "parse" command. Note that the format of the "parse" command is independent of the operating system being used.

Parsing a program module is performed by one of the following commands:

parse *filename*.ffp [-options]	for VAX Computers
vp *filename*.ffp [-options]	for PCs

where *filename* is the file name of the user program module. The file name is arbitrary, but it must have ".ffp" as the file type. Optional additional parameters may be specified on the parse command line following the file name.

As a part of the VSP compilation process, several files with the file name *filename* are created. These are:

filename.c	- main output to be processed with the system 'C' compiler.
filename.ph	- initialization "include" file containing VSP instructions.
filename.ck	- "include" file to check size of compile time initialized fields.
filename.sub	- "include" file containing functions generated from instruction blocks.

As an example, assume that the example program in section 9.2 was created with a file name "example.ffp". This program could be parsed by issuing the command:

parse example.ffp	on the VAX, or
vp example	on the PC.

Note in this case that no optional parameters were specified on the "parse" command line.

9.5 Compiling, Linking and Running a Parsed Program

After the parsing process is complete, the output file "filename.c" must be compiled by the system 'C' compiler and then linked with the VSPS. The commands to do this are dependent on the operating system and are described in sections 9.5.1, 9.5.2 and 9.5.3 for ULTRIX, VMS and DOS respectively.

9.5.1 Compiling and Linking Under ULTRIX

Compiling a 'C' source program under ULTRIX is implemented as:

```
cc -c filename.c
```

'cc' calls the system 'C' compiler with the input file *filename*.c. The '-c' option suppresses loading the program after compilation is complete. The output from the compiler is the object file '*filename*.o', which can then be loaded along with the simulator libraries using the command:

```
f77 filename.o /sim_path/ffp.a -lm -o filename
```

f77 is a shell script in ULTRIX which calls the loader. f77 is used because the VSP simulator contains the IEEE routines which are written in Fortran and call Fortran libraries, while the main body of the simulator is written in 'C'; f77 is thus used to link both the Fortran and 'C' libraries. This load produces an executable copy of the simulator with the name *filename* containing the user source code.

The simulator containing a linked copy of the user source code is run by typing '*filename*' <cr> at an ULTRIX prompt.

9.5.2 Compiling and Linking Under VMS

Compilation is implemented under VMS by issuing the following command:

```
CC filename.C
```

This compiles *filename*.C, and creates the object file *filename*.O. This output is loaded along with the simulator libraries using the command:

```
LINK filename.OBJ, FFP$INC:EMAIN, FFP/LIB
```

This produces an executable copy of the simulator named *filename*, with a link to the user program module. To execute the linked program, type 'run *filename*' <cr> at a VMS prompt. The logical FFP$INC should be defined in the local user login.com file or by the system to point to the appropriate directory.

9.5.3 Compiling and Linking Under DOS

When the DOS diskettes are distributed with a copy of the simulator, four batch (.BAT) files are included which simplify the compiling and linking process. The names of the four files are descriptive of the functions performed by them. The four files are:

vp.bat	for parsing user ".ffp" programs
vc.bat	for compiling ".c" files from the parser output.
vl.bat	for linking the compiled program to the simulator.
create.bat	for combining the three commands above into one

In order to manually parse, compile and link user programs, it is only necessary to issue the commands as shown below:

vp *filename*.ffp [-options]
vc *filename*.c
vl *filename*

After the 'vl *filename*' command is issued, the simulator is ready to run by issuing a '*filename*' <cr> command at a DOS prompt.

The fourth file listed, 'create.bat', can be used to automate the manual parse, compile and link process. For instance, the command:

create *filename*

will parse an input file of the name '*filename*.ffp', compile the '*filename*.c' output file from the parser, and create the executable file named '*filename*.exe'. The simulator with a copy of the user source program is ready to run by issuing '*filename*' <cr> at the DOS prompt.

9.5.4 Running the Compiled Program

Following the steps described above, a copy of the VSPS containing the linked program module is now running. The *Main* Menu of the VSPS will appear on the terminal screen. To run the program, select option '8' from the *Main* Menu, identified as:

"execute user code in vspop()".

The simulator will branch to and execute the code written in the original source program and contained in the function vspop(). When execution is completed, the simulator will return operation to the *Main* Menu.

All of the menu functions, utilities and other tools discussed in Chapter IV are still available for system and algorithm development and debugging when user programs are linked to the simulator. The only difference between the simulator now running and the simulator which runs without linking the user source code is that option '8' from the *Main* Menu now has a vspop() function from which to execute a user program.

The example in Section 9.2 loads simulated external VSP memory with specific values created from the 'C' program. If desired, this part of the user source code could be eliminated. The memory space could instead be loaded manually from the signal generator within the simulator, or loaded from an external signal library. After the memory is loaded, then option '8' could be run which would perform the FFT on the data in that memory. Additionally, the VSPS can be used to plot simulated external memory and examine internal memory and register contents. Any other option in the VSPS menu structure can be selected before or after running the user program. If the message level in the simulator is defined as either 2 or 3, any option in Menu M-4 can be selected for memory or register display prior to or after each instruction is executed.

9.6 Linking Multiple Applications Programs to One VSP Simulator

The way the VSP simulator has been presented, each time an application is written for the VSPS, a user source file is created, parsed, compiled and linked into an executable module. If a number of independent applications requiring the VSPS exist, each of them would individually link a copy of the simulator creating multiple large executable files, one for each application.

The way around creating multiple copies of the simulator is to create a user menu within the main vspop() function which in turn calls different user applications, each written as an external function call. If a new application is generated, simply add its name to the menu called from vspop() and add an external function call to the new application. Note that the individual applications should not have a vspop() function; this should be reserved for the main routine containing the calling menu.

The new application source file is parsed and compiled in the same manner as described in sections 9.5.1 through 9.5.3 for single applications which are linked to the simulator. However, when linking multiple applications to one simulator, all of the application object modules must be specified on the link command line to create the final executable file which is able to call all of the user applications.

When linking under ULTRIX and VMS, this is a straightforward task of simply adding the object file for each application to the link command line. Under DOS, it is a matter of creating a *filename*.lnk file which simply contains the names of the user modules which are to be linked to the application. Additional discussions of this process under DOS are contained in Appendix A.3.

9.7 Using Fortran for Simulating the External Environment

Although the VSPS parser translates code written in ‘C’ with embedded VSP instructions into a ‘C’ program, it is not necessary to learn the full constructs of ‘C’ to program the VSPS. Any language that can be called from ‘C’ can be used to write the bulk of the simulation for the external environment. There are a few constructs about ‘C’ that must be understood in order to properly set up the main program. These include properly defining the beginning and end of the program, as well as the correct format for calling subroutines and passing parameters from ‘C’ to the language performing the bulk of the simulation. Unfortunately, the conventions for calling subroutines and passing parameters vary with different operating systems and languages.

This section describes the minimum ‘C’ knowledge required to call Fortran subroutines and pass parameters under the ULTRIX, VMS and DOS operating systems. Familiarity is assumed with the example in section 9.2 and the ‘C’ constructs described in section 9.3. Variable and array declarations in ‘C’, as well as assignment statements and the ‘C’ "for" statement (equivalent to the Fortran "DO") are covered in Appendix C for those not familiar with ‘C’ programming basics. An example ‘C’ program which calls Fortran subroutines is included at the end of this section, and on the disks which accompany the PC version of the simulator. It may be helpful to refer to this example when reading sections 9.7.1 through 9.7.3.

9.7.1 Calling Fortran Subroutines From ‘C’

Calling Fortran subroutines from ‘C’ requires knowledge of the conventions for naming subroutines and passing parameters. These are not the same in all three operating systems currently supported by the simulator. Under ULTRIX, the underscore character "_" must be appended to the name of any Fortran subroutine. This must not be done under either VMS or DOS. Also, the conventions for default declaration of integer values are different between the two operating systems. Under ULTRIX, an integer value defaults to 32 bits or INTEGER*4. Under VMS, the default is to 16 bits or INTEGER*2. The remainder of this section applies to all operating systems.

In Fortran, parameters are passed to subroutines by *address*. In ‘C’, parameters are normally passed by *value*. When a parameter is passed from‘C’ to Fortran, *the ‘C’ construct must be used which passes the address of the parameter* to maintain compatibility with the Fortran subroutine. The variable name should be prefaced with the ampersand, "&", and enclosed in parentheses, as can be observed in the ‘C’ program example at the end of this section by the

```
initmem_(&(EXTMEM[0]))
```

command line. "initmem_" is the name of the Fortran subroutine being called (note the underscore in the subroutine call used for ULTRIX), while the ampersand passes the *address* of the first element in the EXTMEM array to the Fortran subroutine. It is not possible to pass literals or expressions in this manner;

instead, the literal or expression must first be assigned to a variable and then the address of the variable is passed to the subroutine.

For the sake of accuracy, in 'C' arrays are referenced through pointers to structures. However, it is not necessary to understand either pointers or structures to correctly pass arrays to Fortran subroutines. The example just discussed illustrates the concept of passing arrays to Fortran subroutines.

All arrays in 'C' start at index 0. In Fortran, the common convention (and the only one supported by some compilers) is to start the array at index 1. This should be kept in mind when passing parameters between 'C' and Fortran.

The following example is functionally identical to the example written with combined 'C' and VSP instructions in Section 9.2; it illustrates all the points elucidated in the above paragraphs. The example is designed to work under ULTRIX, VMS or DOS. This requires that different names be used depending on the operating system which is being used. To allow for this, the program uses conditional compilation in 'C'. The code between "#ifdef unix" and "#else" will be compiled only under ULTRIX, whereas the code between "#else" and "#endif" will be compiled only under VMS.

```
C                              Program = exsub.f
C
C     Fortran example file simulating the environment external to the VSP.  These
C     subroutines are called from the vspop() routine in the main 'C' program.
C     These subroutines provide the same function as the 'C' code simulating the
C     external world in the example of section 9.2.

SUBROUTINE INITMEM(IEXTMEM)
      INTEGER*2 IEXTMEM(1)
      DO 100 I=1,64
      IX=2*(I-1)+1
      IF (I.LT.33) IEXTMEM(IX) = 16384
      IF (I.GE.33) IEXTMEM(IX) = -16384
      IEXTMEM(IX+1) = 0
100   CONTINUE
END

SUBROUTINE OUTCMPLX(IEXTMEM)
      INTEGER*2 IEXTMEM(1)
      DO 100 I=1,64
      IX=2*(I-1)+129
      PRINT 50, I-1, IEXTMEM(IX), I-1, IEXTMEM(IX+1)
50    FORMAT("real(",I6,")=",I6,",imag(",I6,")=",I6)
100   CONTINUE
      PRINT 60, IEXTMEM(513)
60    FORMAT ("scale register= ",I8)
END

SUBROUTINE OUTPWR(IEXTMEM)
      INTEGER*2 IEXTMEM(1)
      DO 100 I=1,64
      IX=2*(I-1)+257
      PRINT 50, I-1, IEXTMEM(IX)
50    FORMAT("magnitude of bin",I6," = ",I6)
100   CONTINUE
C THIS FINAL PRINT NEEDED TO FORCE OUT THE LAST LINE ON VMS
      PRINT 150
150   FORMAT("")
END
```

```
/*                              Program = example.ffp          */
/*  Program example showing how the main 'C' program can call Fortran subroutines and pass parameters to those
    routines.  The program results are exacly the same as those shown in the example of section 9.2.

#include"ffp.h"
#define EXTMEM zrmexter->exmem
/#D#/

vspop()
{
/*  Define the default parameters to be assigned for all instructions.  If a parameter is not specified on the instruction line,
    the default values will be used.  If a parameter is specified both on the command line and in the defaults, the parameter
    on the command line  takees highest priority.
*/

/#
        DEFAULT NMPT:64,RS:0,MDF:3,INTRP:0,ZP:0,MBS:128,MSS:128;
        DEFAULT ZR:0,AD:0,EI:0,RV:0,MGSQ.ADF:2;
        DEFAULT FFT.RBA=0,AS:0,R:0,I:0;
#/

/#E#/

/*  Generate the test signal in 'C' and store it in simulated external memory.  The signal is one cycle of a real square wave
    stored in simulated external memory at indices 0 to 63.  The even indices are the real part of the signal, and the odd
    indices are the imaginary part.  Note that a conditional compilation is being performed depending on whether the
    program is running under ULTRIX or VMS.
*/

#ifdef ultrix
        initmem_(&(EXTMEM[0]));
#else
        INITMEM(&(EXTMEM[0]));
#endif

/*  Using VSP instructions, load the entire 64-point vector into internal VSP memory, take the FFT, and store the results
    back out to external memory.  The STI instruction stores the scale value calculated during the FFT out to address 512.
*/

/#
        LD MBA=0;
        FFT NMBT:64, FSIZ:64, FPS:32, LPS:1;
        ST RV:1, MBA=128;
        STI NMPT:1, OR=1, STR:5, MBA=512;
#/
/*  Using Fortran, display the results of the FFT instruction stored in simulated external memory.  Also display the scale
    register result stored in external memory.  Note that a conditional compilation is performed depending on whether the
    operating system is ULTRIX or VMS.
*/

#ifdef ultrix
        outcmplx_(&(EXTMEM[0]));
#else
        OUTCMPLX(&(EXTMEM[0]));
#endif
/*  Now find power using VSP instructions and store the results in external memory.  */
/#
        MGSQ;
        ST RV:1, MBA=256;
#/
/*  Print this result out using a Fortran subroutine.  */

#ifdef ultrix
        outpwr_(&EXTMEM[0]);
#else
        OUTPWR(&EXTMEM[0]);
#endif

}
/##
```

9.7.2 Compiling and Linking Fortran Subroutines Under ULTRIX

A description of how to compile and link the Fortran and 'C' programs together is illustrated best by example. Assume the two example files above are called "exsub.f" and "example.ffp" as shown in the bodies of the programs. The commands to parse, compile and link them with the VSP simulator are as follows:

```
parse example.ffp
f77 -lm example.c exsub.f ffp.a -o example
```

The first statement invokes the VSPS parser to translate example.ffp to a compilable 'C' program as described in earlier sections. The second statement compiles the resulting 'C' code from the parser output ("example.c"), compiles the Fortran code ("exsub.f"), and links these object modules to the simulator. The executable file "example" is generated as output.

9.7.3 Compiling and Linking Fortran Subroutines Under VMS

Assume the two files above are called EXSUB.F and EXAMPLE.FFP as shown in the bodies of the programs. The commands to parse, compile and link the programs with the VSP simulator are as follows:

```
PARSE EXAMPLE.FFP
CC EXAMPLE.C
FORTRAN EXSUB.F
LINK EXAMPLE, EXSUB, FFP$INC:EMAIN, FFP/LIB
```

The first statement invokes the VSPS parser to translate EXAMPLE.FFP to a 'C' compilable program as described in previous sections. The next statement compiles the resulting 'C' program ("example.c") from the parser output. Then the Fortran program is compiled. Finally, all of the object modules are linked with the VSP simulator and the executable file EXAMPLE.EXE is generated.

9.7.4 Compiling and Linking Fortran Subroutines Under DOS

Assume that the main 'C' program is named example.ffp on the PC. This program calls external Fortran subroutines existing in the exsub.for file. The compilation and linking of the programs is illustrated below as:

```
vp example
vc example
vf exsub
vl example
```

The *vp* command parses the ".ffp" program which in turn generates a ".c" output file. The *vc* command compiles the ".c" output from the parser. The *vf* command compiles the Fortran subroutines contained in *exsub.for*. The *vl* command links the multiple module program together by looking for an *example.lnk* file which contains the linking instructions. The file name of the ".lnk" file should have the same name as the program which contains the vspop() function. In the current example, the *example.lnk* file simply contains the single line of text:

```
FILE example exsub
```

This text tells the linker to link the two separate object modules *example.obj* and *exsub.obj* together.

Note that in cases not using multiple object modules, the *example.lnk* file need not exist at all.

Additional discussions of multiple-module linking under DOS is included in Appendix A-3.

9.8 Additional VSP Simulator Programming Concepts

As the first few sections in this chapter illustrate, the VSP simulator is an easy tool to use for system development. The process of writing VSP instructions and including high-level 'C' or Fortran routines with the VSP instructions is straightforward. However, there are additional powerful programming constructs provided by the VSPS which allow even higher levels of programming sophistication. It is the purpose of the remainder of the sections in this chapter to introduce these concepts. Many of the discussions contained in the remainder of this chapter are also covered with a formal language description in Appendix D.

9.8.1 VSP Statements

The VSP language has three types of statements: VSP instructions, pseudo-operations (commands to the parser and run-time interpreter), and instruction blocks. With the latter, instruction parameters can be efficiently defined outside of instruction loops. Instructions can be labeled so that the program will WAIT until a specified instruction has executed. In a multi-VSP system, instructions can be sent to a designated VSP by following the instruction with the "@ expression" statement. The expression is evaluated to determine the VSP number.

9.8.2 VSP Instructions

The format of VSP instructions used when writing programs for the VSPS is the same as it is in the instruction tutorial mode. For example, the following FFT instruction:

```
FFT NMBT:64,RS=0,ADF=3,AS=0,R:0,FSIZ:64,EI=0,FPS:32, LPS:1, I:0,RBA:0;
```

Parameters can be declared with either LITERAL or LOGICAL values. The designators show whether a parameter value is logical ":" or literal "=". Literal values are the exact value of the parameter which will be used in the instruction field of the instruction at execution time. Logical parameters use a more natural representation of the value for some parameters. Translations from logical to literal values are provided with the description of the instructions in the instruction tutorial section of the VSPS (Menu M-2-InOp).

9.8.3 Pseudo-Operations

Pseudo-operations are commands to the VSPS parser. They provide information to the parser (STATIC and DEFAULT), control VSP/host coordination (WAIT), control the disposition of VSP instructions (DEFER and IMMEDIATE), and control parser options (OPTION).

STATIC: This informs the parser that certain variables are defined at compile or load time, and can be initialized. If a parameter is set to an expression with a variable not declared as static, that parameter will be initialized at execution time.

DEFAULT: This provides default values for instruction parameters to the parser. Default values can be defined for every occurrence of a parameter, or can be set for a specific instruction. The command:

```
DEFAULT NMPT=5;
```

will set the value of NMPT in every instruction using this parameter. The command:

```
DEFAULT LD.NMPT=5;
```

will set the default value of NMPT only in the LD instruction. Similarly, parameters can also be defined as either literal values, using "=", or logical values, using ":".

WAIT: This causes 'C' code execution to be suspended until the specified VSP instruction is completed. DONE suspends execution until all issued VSP instructions have been executed. Specifying a label effects a WAIT for the labeled instruction to terminate. Following a label with a ":" and an expression with the value N suspends execution until the Nth instance of the labeled instruction has been executed. For example, the command :

```
WAIT STINS:4 @2;
```

will effect a WAIT in VSP 2 until the instruction labeled STINS has been executed four times.

OPTION: This statement sets parser options. SET and CLEAR (non VALUE) options can also be set at parse time. For instance, the command:

parse *myprog.ffp* -ccksize -cline

will parse the program named *myprog.ffp* and include the "ccksize" and "cline" options with the parse process. The two parameters need not be specified in the ".ffp" program when specified on the "parse" command line.

All parser options are shown below. Initial values are shown in parentheses.

OPTION	SET/CLEAR/VALUE	MEANING
VSPID	VALUE (0)	Sets default VSP number
ERRLEV	VALUE (0)	Sets level of error messages (no effect in current implementation)
WAITALL	VALUE (0)	1 indicates wait for completion after each VSP instruction
CKSIZE	VALUE (1)	1 indicates check size of all parameters
SCKSIZE	SET	Sets CKSIZE to 1
CCKSIZE	CLEAR	Clears CKSIZE
LINE	SET (SET)	Outputs comments containing source code line numbers
CLINE	CLEAR	Clears line
CMPEQ	SET (CLEAR)	Complains about LITERAL parameter settings
CCMPEQ	CLEAR	Clears CMPEQ
CMPCO	SET (CLEAR)	Complains about LOGICAL parameter settings
CCMPCO	CLEAR	Clears CMPCO
CMPALL	SET (SET)	Complains about every instance for CMPEQ and CMPCO
CCMPALL	CLEAR	Clears CMPALL so only the first instance generates a complaint
INDL	SET (SET)	Causes emission of "#line" card in ".C" output file
CINDL	CLEAR	Clears INDL
LOGICAL	SET (SET)	Use logical parameter values in error messages
LITERAL	CLEAR	Use literal parameter values in error messages

9.8.4 DEFER and IMMEDIATE Programming Constructs

The DEFER and IMMEDIATE pseudo-operations are important commands which require their own section. They provide a number of powerful features when writing VSP programs. They must be used as a pair; a DEFER command requires an IMMEDIATE command at some later point in the program. DEFER causes instructions which follow to be written to simulated external VSP memory at the address specified in the following "expression". The instructions **are not** executed as they are written to external memory. IMMEDIATE turns off DEFER and causes subsequent instructions to be executed immediately.

In "real" hardware VSP applications, the VSP would normally fetch its instructions out of external memory and then execute those instructions. When programs such as the example program in section 9.2 are run, the simulator makes no attempt to fetch instructions out of simulated external memory as would be done in a "real" system. The instructions are executed within the simulator, but no attempt has been made to model the storage and fetching of instructions out of external memory.

The DEFER instruction writes all instructions which follow into simulated external memory at the address specified by the "expression" until an IMMEDIATE command is encountered. The instructions are written to memory in the binary image that a hardware VSP would expect to find in a "real" system. The reason for doing this is that now the simulator can actually model the VSP fetching instructions from external memory. Additional parameters such as bus timing and usage as well as the number of clock cycles required can also be accurately modeled.

A second important feature is provided by the DEFER/IMMEDIATE commands. When instructions are written (DEFERred) to simulated external memory, they are written in the same binary executable format that a hardware VSP would expect to read in a "real" system. The simulator is able to dump this simulated memory containing binary executable VSP instructions in a JEDEC format to a disk file. The JEDEC file is in the proper format for burning into PROMs using industry-standard programming tools. The PROMs can be used directly in a hardware system incorporating the VSP. The VSP simulator then ties together the instructions executed in the simulator environment with the formatting for the end hardware system.

The following example shows how the DEFER and IMMEDIATE commands are used.

```
#include "ffp.h"
#define EXTMEM zrmexter->exmem
/#D#/
vspop()

/#E#/
        EXTMEM[subroutine] = base;
/#

/*   Note: no VSP instructions will be executed after the DEFER statement until an IMMEDIATE statement is
     encountered. */

        DEFER base;
               /* VSP code to be DEFERred */
        HLT;
        IMMEDIATE;
#/

/*   JMPI begins execution of the DEFERred code; HLT ends program execution.        */

        JMPI subroutine;
```

When this program is run, all VSP code between the DEFER and IMMEDIATE commands will be written to external memory, including the HLT. Note that JMPI is not written to external memory because it is outside the DEFER/IMMEDIATE loop. When the JMPI is encountered after the IMMEDIATE command, the simulator jumps indirectly to the beginning of the instructions which exist in simulated external memory at address "base". All instructions are executed from external memory until the HLT instruction is encountered.

Instructions which have been DEFERred cannot be controlled by the WAIT statement. If DEFER and IMMEDIATE options are used, coordination between the VSP and host must be handled directly, or else reliance must be made on the "WAIT DONE;" statement. Chapter X discusses host and VSP coordination in more detail.

9.8.5 Creation of JEDEC-Formatted PROM Files

If the instructions execute correctly in the simulator, users may want to create JEDEC formatted files of these instructions for burning into PROMs. Option '14' from Menu M-4-20 is used for this purpose. This selection will prompt for the file name to write the JEDEC formatted instructions into, as well as the number of words of memory and the starting address in simulated external memory where the instructions exist.

The simulator currently assumes that the PROMs to be burned are of size 2K words by eight bits (16K). The JEDEC-output file name is specified by the user. The simulator appends a number to the end of the user-specified name corresponding to the number of 2K-word blocks which are created. In addition, the formatter assumes that the memory size is 16 bits (as in the simulator), so high and low byte files are created. For instance, if 3K words of instructions exist in simulated external VSP memory and are formatted by the JEDEC option, the program will create four files, named *file*0.lo, *file*0.hi, *file*1.lo, and *file*1.hi, where *file* is the name provided by the user.

9.8.6 Instruction Blocks

The instruction block construct supported by the simulator can be useful for filling in instruction parameter fields for the VSP when repetitive loops are being implemented. For instance, the overlapped I/O example discussed in section 8.2.2 introduced the concept of the FFT loop. With the basic instructions of ST, LD and FFT, FFTs of various sizes can be implemented using the 64-point FFT as a building block. While this example did not specifically address the implementation of large FFTs, the repetitive nature of the loop was illustrated between instructions 3 and 9.

The BUILD construct of the simulator is designed to simplify the process of instruction parameter field definition for the specific case when nested instruction loops exist and some parameters in the outer loop change slowly while parameters in the inner loop change quickly.

The syntax for instruction blocks is similar to that for 'C' subroutines. The instruction delimiters '/#' and '#/' can be used inside instruction blocks; 'C' code placed between them is included in the subroutine created for the block. Each instruction block defines a 'C' subroutine of the same name and with the same set of parameters (all of type "int") as the instruction block name and parameters.

Code for an instruction block is emitted at two distinct places. The first is within the subroutine, and the second is at the point in the calling program where the block is defined. The subroutine definition is written to a separate file to be read with "#include" where the "/#D#/" marker occurs. All fields that do not contain parameters for the block will be filled in when the block definition is executed. The remaining parameters will be filled in when the 'C' subroutine with the same name as the block is called.

The code to execute VSP instructions is issued when the subroutine is called. Because of this, the block definition must be executed before the call to the instruction block to fill in the fields. The example below shows how an instruction block might be used.

```
/*          Example: Static Declaration and Instruction Block Build*/

#include "ffp.h"
#define offset 62
/#D#/
vspop()
{
            int i, j, ipb, base, base2;
/#
            DEFAULT NMPT:64, FFT.NMBT:128, FFT.RBA=0, INTRP=0, ZP=0,
                        MDF=3, ADF=3, RS=0, EI=0, MBS:1, MSS:2, RV=0, ZR=0,
                        AD=0, SH=1, STR=1, AS=0, R=0, FSIZ=4, FPS=0, LPS=6, I=0,
                        SCL.SB=1;
            DEFAULT LDSM.NMPT=1, LDSM.UP=0, STI.OR=1;
            STATIC offset;
#/

/#E#/
            for (i=1; i<100; i++) {
/#
                        BUILD seti(ipb) {
                                    LD MBA:ipb;

/*   MBA in this instruction is a function of the block parameter ipb which is passed when the "seti" function is called. It
     must be passed each time the block is called, or 10,000 times per execution of the outer "for" loop.
*/
                                    ST MBA:base2+i;

/*   MBA in this instruction is calculated for each value of i, or 100 times per execution of the outer "for" loop.
*/
                                    STB MBAB:offset;

/*   The BUILD procedure sets all instruction fields in block "seti" that are a function of i.
*/
                        }           /* end of the BUILD procedure */
#/
                        for (j=0; j<100; j++)
                                    seti(j + base);
            }           /* end of the loop on i. */
}
##
```

The operation of this program is as follows. The BUILD construct names a function "seti" which defines three instructions; in this instance they are LD, ST and STB. The BUILD definition is executed 100 times, but the block of code associated with the BUILD construct is not executed in the outer "for" loop; it is only executed when called by "seti" in the inner "for" loop. Each time BUILD is executed in the outer loop, the parameters in the ST and STB instruction are updated. For each time BUILD is executed, the inner loop calls "seti" 100 times and passes the parameter "ipb" to "seti". Therefore, the parameter MBA in the LD instruction is updated 10,000 times because "ipb" is passed to the "seti" function 10,000 times. However, the parameter MBA in the ST instruction is only updated 100 times - when the BUILD definition is executed. The parameter MBAB in the STB instruction is never changed because it is defined as the value 62 by the "#define" at the beginning of the program.

The power of the instruction block construct is the ability to set slowly changing parameter values in an outer loop rather than in an inner loop. This reduces the overhead involved in always defining parameter fields in an inner loop; in large applications execution time may be increased significantly.

9.8.7 Programming Hints

- It is preferable to use logical parameter values for instruction fields in most instances. This makes it easier to understand the fields in the instruction.

- When the MODE register is set, all parameters in the register are changed. When changing a single parameter, ensure that each parameter is set to the desired value.

- The VSPS programs expect to operate with two RAM sections; the simulator should be left in this mode when control is returned to the interactive mode.

- In linking with the VSP simulator there may be name conflicts between names used in "FFP.H" and VSPS external variables. Compiler or linker errors may be received due to this.

CHAPTER X

SIMULATING THE ENTIRE VSP/HOST SYSTEM

10.1 Overview

The VSP simulator executes VSP instructions with the assumption of a specific hardware architecture and hardware configuration which is described in this chapter. The parser also must address a specific form of the assumed host controller operating system which deals with the queueing of VSP instructions to be executed.

In general, VSP instructions are not executed as soon as they are encountered in a user program; they are entered in an instruction queue for later execution. The VSPS simulates two types of software queues that work together with the VSP on-board FIFO. The software operating system manages these software queues. This can be done using polling methods or using special interrupts from the VSP.

10.2 Hardware Configuration

The VSPS models up to eight VSPs in parallel in a single application, all sharing the same host controller, RAM and bus. This configuration is shown in Figure 10-1.

```
 ______     __________     ______     ______             ______
| HOST |   | EXTERNAL |   | VSP0 |   | VSP1 |  .....    | VSP7 |
|______|   |___RAM____|   |______|   |______|           |______|
    |           |             |          |                  |
    |           |             |          |                  |
    |___________|_____________|__________|__________________|

                        DATA AND ADDRESS BUS
```

Figure 10-1. Hardware architecture assume by the VSPS.

Since the bus is shared, if more than one VSP is accessing the bus at any one time, all but the highest priority VSP will have to wait to gain control of the bus. Priority among VSPs is assigned according to their index number. VSP 0 will get the bus before VSP 1 if they both request it in the same bus cycle. Delays encountered in waiting for the bus when there are multiple VSPs in the system are modeled by the simulator.

The VSPS monitors the number of bus cycles used in executing user programs. The number of clock cycles can be monitored by using menu M-4-21 which selects the VSP clock sub-menu.

A subroutine to be described later can be used to model the usage of the bus by the host controller.

10.3 Instruction Queueing in General

Since the VSP normally operates independently of the host controller or microprocessor, the execution speed of instructions can be increased by queueing instructions for the VSP. The host can be freed for other processing if it can dump instructions for the VSP into a queue, then return once the VSP has completed their execution. There are a variety of possible approaches to instruction queueing with different cost/benefit tradeoffs. The simplest scheme that uses the least host memory is the hardware queueing of up to four instructions built into the VSP using the on-board FIFO. The hardware queue may be supplemented with software queues, allowing the host to dump larger blocks of VSP instructions into an external memory buffer. The memory buffer is then either read by the VSP, or the host later transfers instructions from the software queue into the VSP instruction FIFO. The efficiency of software queues may be improved by allowing the VSP to fetch its own instructions as well as by using the JMPI and HLT instructions for moving between blocks of instructions within queues.

Figure 10-2 shows the relationship between these different queues. The dividing line separates the hardware queue (maintained entirely within the VSP by the host), from the software queue, which is maintained entirely by the host processor. The fetch queue resides midway between these two methods. The fetch queue uses host external memory for instruction queueing, and the VSP and the host jointly manage the instructions in the queue. Additional discussion on VSP instruction queueing is handled in the sections to follow.

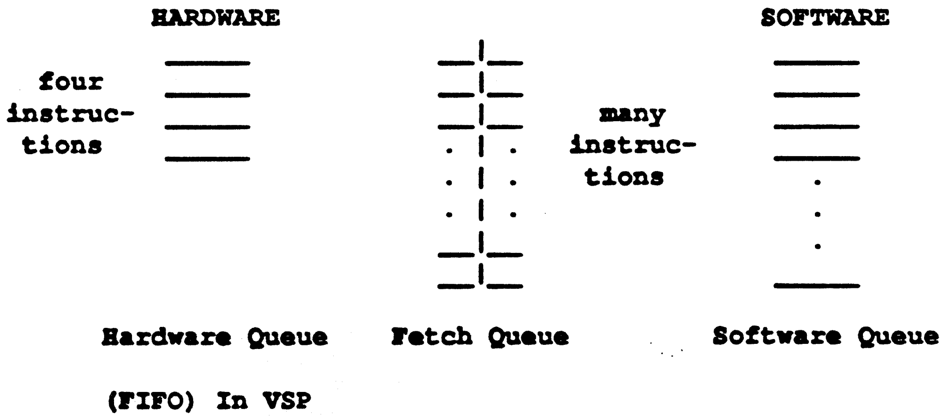


Figure 10-2. Relationship of Queues

The inter-dependence of the host controller and VSP requires that the operating system have a way to control the flow of execution. The WAIT statement in the VSPS programming language meets this need. It stops the host from issuing further instructions to the VSP until it has executed all the instructions in the queues. There are many ways to use the WAIT statement: it can be issued in an instruction stream, or it can be issued to operate on a certain occurrence of an instruction during a loop. This will be described more fully in Appendix D.

The major alternatives for instruction queueing and their relative advantages are listed below.

10.3.1 Hardware Queueing Using the VSP Instruction FIFO

The first and simplest approach for queueing instructions to the VSP is to use only the hardware queue (instruction FIFO) in the VSP. When an instruction is to be executed, the host checks the FIFOSTAT field of the status register. If an instruction slot is open, the instruction is written to the VSP. If no slot is available, the host can either loop, continuing to check for an available slot, or wait for an interrupt from the VSP. When an instruction slot is available within the VSP, the host writes the instruction into the instruction FIFO.

10.3.2 Hardware and Software Queueing

A step beyond using just the VSP instruction FIFO for queueing is to use a software queue from which instructions are written to the VSP instruction FIFO by the host controller. A system using a software queue first checks to see if the hardware queue of the VSP is full. If it is not full, the instruction is written into the VSP FIFO. If the hardware queue is full, instructions are placed in the software queue in system memory. The host must continuously check the VSP instruction FIFO to see when space is available to move instructions from the software queue to the FIFO.

While adding a software queue is an improvement over simple hardware queueing, there is still significant host overhead associated with polling the FIFOSTAT register and handling an interrupt when the queue is empty.

10.3.3 Instruction Fetch Queueing

One way to get around the disadvantages of the two methods just proposed is to use the instruction fetch feature of the VSP for instruction queueing. VSP instructions are written to a block of memory by the host controller which is common to both the controller and the VSP. Then VSP instruction fetch is started from that common block. While the VSP is fetching instructions from one portion of the common queue, the host controller can be writing additional instructions into another part of the queue. Note that this method differs from the software queue in that the VSP fetches its own instructions from the queue. In the software queue example, the host wrote instructions either to the VSP or the software queue as a function of the FIFOSTAT bit in the VSP.

Any new instructions to be queued are appended to the queue by the host controller. A JMPI instruction at the physical end of the block can be used to return the program to the start of the block, using it as a circular or continuous buffer. A HLT instruction must be used at the logical end of the block so the VSP doesn't execute invalid instructions. When the host has additional instructions to append to the end of this block, it does so by writing the entire block, all except for the first word of the first instruction. Once this is done, the HLT can be overwritten with the first word of the first instruction. Caution should be used with this method as described in section 10.5.

10.4 Interrupt Support for Queueing

There are three types of VSP interrupts which help with instruction queueing. The types of interrupts and their associated bits are:

- IFO, indicating an instruction FIFO overflow.
- ILI, indicating the last instruction in the FIFO has been executed.
- IMI and IAI, one of which will be set after any instruction is executed with the EI (Enable Interrupt) bit set. IMI indicates that some type of data move instruction has completed, while IAI indicates that an arithmetic instruction has completed.

Under some conditions and on some systems it may be more efficient to write an instruction to the hardware queue and test for overflow than to check beforehand for an available slot. This function and error detection are the principle uses of the interrupt on FIFO overflow.

The FIFO-empty interrupt is useful when using a software-queued system. The interrupt informs the host controller that the VSP instruction FIFO is empty and instructions should be written from the software queue to the FIFO. This should be done with caution when using a fetch queue, because the instruction FIFO can become empty when the VSP is in the middle of an instruction fetch block.

Referring back to Chapter VI, it can be observed that every instruction but HLT has an EI parameter in the LSB of the first word in the instruction. This bit can be set to flag the end of an instruction fetch block. The EI bit of each instruction can be used to signal when a particular instruction has been executed. This can operate to control both software and instruction-fetch queueing methods.

10.5 Queueing Algorithms

Instruction queueing algorithms which can be used for programming the VSPS are straightforward. Regardless of the queueing method chosen, the user always has control of the VSP through the proper use of the mode register, the interrupts and the applicable instructions. If a software queue is used, and the VSP instruction FIFO is full, the insertion of additional instructions into the FIFO is not possible until the space is available. As an alternative, the memory space for the software queue can be increased using the "malloc" function in 'C'.

If interrupts are used, the transfer of control between the software queue managed by the controller and instruction fetching managed by the VSP is precise and efficient. If interrupts are not used, the run-time interpreter tests for the need to transfer queue control. There is a special call to the interpreter to request such a check.

When instruction fetch queueing is used, particular attention must be paid to the use of the HLT command. When an instruction is added to the queue, the HLT instruction may be overwritten at the logical end of the queue as the new instruction is added. It may happen that the VSP tries to execute HLT while the host controller is updating the queue. To guard against this, do the following:

1. Save the first word of the first instruction in a different memory space; do not write it into the software queue yet.

2. Write the second word of the first instruction to the second word of the existing HLT instruction. Although HLT is two words, the second word is a DON'T CARE.

3. Write all succeeding instructions after the HLT instruction.

4. Write HLT at the end of these instructions.

5. Now overwrite the first word of the original HLT with the first word of the first instruction to be executed.

6. Read the instruction counter to determine if the instruction fetch unit read the HLT before it was overwritten. If it has not reached the address of the HLT, or has passed it, the HLT was not read.

7. If the instruction counter is at the second word of the instruction following HLT, or at the word after that, the WAIT command must be used until all queued instructions have been executed; then the user must find out if the program counter goes past the address after HLT.

8. If it does, the HLT was not read, and the program can continue. If it doesn't, and the VSP goes idle, the HLT was read, and the proper instructions must be used in order to continue the program.

Quite a bit of information on instruction queueing algorithms can be found in the on-screen text in the various HELP options. Menus M-1 through M-6 all have HELP screens.

10.6 VSPS Instruction Queueing

The software operating system simulated by the VSPS can be set up to use any of the three queueing options described in Section 10.3 above. It can also run using a fourth queueing method which combines the first three.

10.6.1 Enabling VSPS Hardware Queueing

Using this option, the VSP simulator models only the hardware queueing of up to three instructions, as described in section 10.3.1. When an instruction is to be executed from the source program, it is sent to the VSP instruction FIFO. If the queue is full, the host stops sending the instruction and polls the FIFOSTAT parameter of the VSP status register until the queue has a free slot. Keep in mind that this procedure causes the interrupt bits in the VSP to be reset. When the host finds a free slot in the VSP, it writes the next instruction and continues execution.

This model is enabled by selecting option '2', '3' or '4' from Menu M-4-22, then selecting option '8' from the same menu to shut off all software queueing. To disable this model (and all queueing) select option '1' from Menu M-4-22.

This model is also enabled by using the ZRSETSYS subroutine described in Section 10.8.

10.6.2 Enabling VSPS Hardware and Software Queueing

In this case a software queue is used as described in Section 10.3.2. Interrupts are not used, so the VSPS models the host as polling the status register, with the result that interrupt bits are reset every time it is read. This cannot be done through the interactive interface; it can only be enabled by a call to the ZRSETSYS subroutine from the user source program. This is described in Section 10.8.2.

10.6.3 Enabling the VSPS Hardware and Instruction Fetch Queue

This option defines a queueing system as described in Section 10.3.3. Again interrupts are not used, and the VSPS host model polls the status register of the VSPS. This option cannot be reached through the interactive interface, and is enabled only by calling the ZRSETSYS subroutine from the source program. This is described in Section 10.8.2.

10.6.4 Enabling VSPS Fetch, Software and Hardware Queues

With this option, the VSPS combines all of the queueing methods described in Section 10.3 into one method. The reason for providing this method is that when there is considerable VSP/Host interaction, it may not be feasible to queue more than a few instructions before a WAIT is required. The additional overhead of setting up an instruction fetch queue may not be justified by the queue length. A combination of software and instruction fetch queueing techniques can be used to get around this problem.

The combined queue simulation operates as follows: When an instruction is to be executed from the source program, it is written to the VSP if the hardware queue has an available slot. If no such slot is found, a software queue with a pre-determined length is checked, and if a slot is available, the instruction is written to this software queue. If the software queue is full - past a preset size - an instruction fetch queue is set up, instructions are transferred from the software queue to the fetch queue, and the new instruction is entered in the fetch queue. When the hardware queue has a slot available, a JMPI to the instruction fetch block is executed.

The ZRSETSYS subroutine is used to enable this queueing option and to define its parameters.

10.7 Modeling Host/VSP Coordination

When a queueing model is enabled in the VSPS, the way the host and VSP interact on the shared bus can be selected. Of the four choices, the first three determine how long it takes for the VSP to return control of the bus to the host. With the last option, users can define their own model of the host. Figure 10-3 shows the relationship among the first three methods.

```
  Option 2              Option 3                  Option 1
  Control       Control Returned to Host           Control
  Returned       At Random Time In This           Returned
  To Host               Interval                   To Host
   Here                    |                        Here
    |                      |                         |
    |                      |                         |
    |                      v                         |
    v   __________________/\____________________     v
       /                                        \
     /                                            \
  |------------------------------------------------|->
  ^                      TIME                      ^
  |                                                |
Host Issues                                  VSP Completes
Instruction                                   Instruction
```

Figure 10-3. Host/VSP Coordination Modeling Schemes.

10.7.1 Complete Instructions as Issued

With this option, the host waits for the VSP to finish the instruction currently being executed. This option will remove any scheduling problems caused by queues. It is most useful when initially debugging VSP programs and is invoked by selecting option '2' in Menu M-4-22.

10.7.2 Immediate Return to Host

With this option, the host continues execution of code immediately after issuing a VSP instruction. The VSP instructions are executed when some form of the WAIT statement is reached, or when the active queues become full. This option is the opposite of the first method above, "complete instruction as issued" (Section 10.7.1), in that the VSP instructions are delayed as long as possible before they are executed, and is invoked by selecting option '3' in Menu M-4-22.

10.7.3 Random Return to Host

This option more accurately simulates the conditions in a real system. The upper and lower bounds of a uniform random number generator can be set which determines the number of clock cycles elapsing before the VSP returns to the host.

This option is invoked by selecting option '4' in Menu M-4-22.

10.7.4 User Defined Host

The host controller and the number of bus cycles it requires on the bus in executing the VSP program are not specified by the VSPS. The particular bus requirements of the host program can be defined by using the ZRHSTBU subroutine. This subroutine is called every clock cycle. Its form is

```
zrhstbu(busav)
        int busav;
           {
                <your model goes here>
           }
```

The variable "busav" has the value 0 if the bus is not available to the host; if the bus is available, the value of "busav" is 1. A 0 is returned if the user host model requires the bus; a 1 is returned if it doesn't.

10.8 System Setup Subroutine

A 'C' subroutine for defining queueing and other system parameters, ZRSETSYS, is callable from the user program. Most of the options described above can be set using the menus. ZRSETSYS is used to set them from a VSP program.

10.8.1 General Features of ZRSETSYS

The options of ZRSETSYS allow defining the maximum number of VSPs in the system, the maximum size of external memory, and the options that control the software and fetch queues.

With this first set of options, the hardware system can be defined by specifying the number of VSPs and the size of external memory.

MAXVSPS VAL Specifies the maximum number of VSP devices to simulate on the same bus; VAL must not exceed eight. The default value is eight.

MAXEXTME VAL The maximum size of external memory. The maximum and default values are 65536.

The next set of options controls how the host and VSP interact. Section 10.6 describes how to do this using the simulator menu structure.

NOQUEUE Do not model any VSP queueing. This is the default option.

QRETCOMP Model the hardware queue and return control to the host program as soon as an instruction is queued.

QRETIMM Model the hardware queue but do not return control to the host program until each VSP instruction has completed execution.

QRANMIN VAL Model the hardware queue and return control to the host program after a random number of clock cycles. The number is taken from a uniform distribution with the minimum value equal to VAL. The default for VAL is 0.

QRANMAX VAL Model the hardware queue and return control to the host program after a random number of clock cycles has been completed. The number is taken from a uniform distribution with the maximum value equal to VAL. The default for VAL is 1024.

The last set of options control how the queueing is performed. These parameters control the selection of the queue models as described in Section 10.6.

SOFTQSZ VAL The maximum number of instructions in the software queue. If this value is zero then the software queue is not used. The default value is zero.

SOFTQLIM VAL The threshold for the software queue. An attempt will be made to move an instruction out of the software queue when this threshold is reached. The default value is 16.

FETCHQBS VAL The base address in external memory for the fetch queue. The default address is the top of memory less the number of VSPs times 256.

FETCHQSZ VAL The size in words of the fetch queue. This space is divided equally among the number of VSPs specified. The default value is 256 for each VSP.

FETCHQTH VAL The threshold queue size for transferring an instruction from the fetch queue to the hardware queue. The default value is 24.

The declaration in the 'C' program for ZRSETSYS is as follows:

```
zrsetsys(opts)
struct ops
   {    char *zrname;
        long val;
   };
opts [];
```

To use ZRSETSYS, an external or static variable should be declared and initialized as follows:

```
static struct {char *zrname; long val;} zrrst[] =
   {
        {"maxvsps",8},
        {"maxestme",zrexmax},
        {"qretimm",0},
        {"softqsz",30},
        {"spftqlim",15},
        {"fetchqbs",zrexmax-4096},
        {"fetchqsz",128},
        {"fetchqth",8},
        {0,0}
   };
```

Each entry in the structure is the name of the option followed by its value. To set the options, ZRSETSYS is called with a pointer to the initialized array as a parameter. For example, with the array above the call would be:

```
zrsetsys(zrrst);
```

10.8.2 Setup for Various Queueing Options

To model any queueing, the hardware queue (VSP instruction FIFO) must be enabled. This is done by setting one of the variables QRETCOMP, QRETIMM, QRANMIN VAL or QRANMAX VAL. Remember, to enable any of the software queues described below, the hardware queue must also be enabled.

To initialize the system for the hardware and software queuing described in Section 10.6.2, ZRSETSYS is called with FETCHQSZ VAL equal to zero, and SOFTQSZ VAL and SOFTQLIM VAL are set to the desired values.

To enable instruction fetch queueing only, set the SOFTQSZ VAL to zero, and set FETCHQBS VAL and FETCHQSZ VAL to the desired values.

If the instruction fetch and software queues are to be modeled as described in Section 10.6.4, initialize the software queue with SOFTQSZ VAL, SOFTQLIM VAL, and the instruction fetch queue with FETCHQBS VAL and FETCHQSZ VAL. Set FETCHQTH VAL to the desired length of the fetch queue. All that is required to enable this queueing method is to set SOFTQSZ VAL to a value other than zero; if the other values are not set, their default values will be used.

10.9 The ZRCKMSG Subroutine

Like the ZRHSTBU and ZRSETSYS subroutines described in the last section, the subroutine ZRCKMSG can be called from within the source program. This subroutine allows resetting the message level of the VSPS during program execution. Recall that the message level determines the amount of instruction printout during execution. It is useful to call this subroutine at critical points in the program to allow skipping over parts of the program that are not of interest. The formal declaration in the VSP simulator for ZRCKMSG is:

```
zrckmsg(level)
int level;

   {
   };
```

The call to the subroutine from the source program is simply:

```
zrckmsg(level);
```

where *level* is an integer variable. The *level* variable sets the message level, which can be any number from 0 through 3 or 10 through 17 corresponding to the options available in Menu M-4.

Note that external RAM can be plotted by calling this subroutine with level = 10.

10.10 A Summary of the VSP and Host Coordination

The code for the host and VSP is intended for two or more processors. In general, VSP instructions are not executed as soon as a VSP instruction occurs in the host stream. Instead, they are entered in a software or hardware queue for later execution. At various points in a program it is necessary to insure that certain VSP instructions have completed execution before host execution can proceed. From a scheduling standpoint the host and VSPs are in a master-slave relationship. The host controls all scheduling by not issuing VSP instructions until all necessary data is available and certain VSP instructions have completed execution.

To coordinate this, execution must be suspended until the VSP has completed certain instructions. For added flexibility a capability must exist to allow testing whether a specific instruction has completed without necessarily suspending execution.

It is important to distinguish between a line of source code that contains a VSP instruction and an instance of that instruction issued for execution to the VSP. The same source code line may be issued for execution many times within a program. In general, it is desirable to WAIT on a particular instance of an instruction. This may not be the most recently issued instruction. A VSP instruction is issued whenever the host executes the line of code containing it. It must be possible to specify the instruction on which to WAIT, as well as which instance of the instruction - the most recent, or a previous one.

10.10.1 Deferred Instructions

Ordinarily when a VSP instruction is reached in a VSP program it is queued for execution. There are two exceptions: the first involves instruction blocks, the second involves deferred instructions. Both subjects are discussed in Chapter IX. There is a software switch in the run-time interpreter that determines the disposition of instructions. This switch is normally cleared. It can be set by the DEFER pseudo-operation and is cleared by the IMMEDIATE pseudo-operation. When it is set, instructions are written to shared host/VSP memory beginning at the address specified immediately after the DEFER command. These instructions can be executed later by using an immediate JMPI instruction to branch to their location in simulated memory. However, the run-time interpreter does not monitor deferred instructions. When using DEFER, users can define their own queueing and generate data to be used in programming ROMs for the VSP.

10.10.2 Instruction Labels

To assist with queueing, the VSP language uses *instruction* labels and the WAIT and CHECK pseudo-operations. Instruction labels are similar to those found in 'C'. They are stored in a separate table. The same name can be used to label 'C' statements, but this is not generally good practice. WAIT and CHECK statements can refer to any labeled instruction. VSP instruction labels must be unique in each separately compiled program; the VSP parser does not use the 'C' block structure in limiting the scope of labels.

10.10.3 WAIT Statements

The WAIT statement is used in two ways for a single VSP: WAIT on an instruction, and WAIT DONE. WAIT ALLDONE effects a wait until all VSPs have completed execution.

WAIT on an Instruction

```
WAIT label[ :expression][ @expression];
```

Any valid variable name may be used for "label". The label and the WAIT statement must be in the same compiled module. The ":expression" part is optional; if it is omitted, the WAIT will apply to the most recently issued instance of the labeled instruction. Otherwise the expression is evaluated each time the WAIT statement is executed, yielding an integer value N. The WAIT will then be on the Nth most recently issued instance of the labeled instruction. If N is 0, the WAIT is on the most recent instance. If fewer then N instances of the instruction have been issued, no WAIT occurs. This is not flagged as an error. Negative N is a fatal execution time error. Limitations on queue size restrict values of N to a few hundred. The "@ expression" part is also optional. It designates the particular VSP to which the statement applies.

WAIT DONE

```
WAIT DONE[ @ expression];
```

The optional "@ expression" designates a particular VSP. If this part is omitted, the statement applies to the default VSP. This statement waits until all instructions issued to the specified VSP have completed execution. This form of WAIT can be used with deferred instructions.

WAIT ALL DONE

This statement causes a wait until all instructions to all VSPs have completed execution.

CHECK and SET

```
CHECK ALL DONE SET name;
CHECK SET name : expression : expression[ @ expression]
CHECK SET name expression;
CHECK SET name DONE @ expression
```

The optional "@ expression" part has the same interpretation as in the WAIT statement. The "name" after SET is the name of an integer variable declared in the 'C' program. The CHECK and SET statement sets the named variable to 1 if the labeled instruction has completed execution. Otherwise the named variable is set to 0.

APPENDIX A

VSPS INSTALLATION PROCEDURE

The installation procedures for the VSPS are different under all of the three operating systems supported by the simulator: VMS, ULTRIX and DOS. A summary of the installation procedures is given for all of the operating systems in the following sections.

VMS and ULTRIX differ in the way they treat upper case characters. ULTRIX always distinguishes between upper and lower case. VMS accepts file names and program names in either upper or lower case and always outputs such names in upper case. For this reason all VMS examples are shown with names in upper case. When using VMS, either upper or lower case may be used for entering text. In contrast, examples under ULTRIX use mostly lower case and the correct case must be used in entering the names under this operating system.

A.1 VSPS Installation Under VMS

A.1.1 General Information

You should select a directory in which the VSP software is to be installed. All the files needed for executing the VSP simulator should be put in this directory. VMS accepts file names and program names in either upper or lower case and always outputs such names in upper case. For this reason we give all VMS examples with these names in upper case. You may use either upper or lower case in entering them. In writing your own VSP programs you must link your program with the VSPS simulator. This can be done in two ways. Either you can create a totally independent executable module, or you can create a sharable image. The former approach is more portable between different VMS versions. The sharable image approach uses less disk space and links much faster. We provide instructions for both types of installation.

The steps for installing the VSP simulator and language under VMS are as follows:

A.1.2 Reading the Distribution Tape

1) Define the physical name of the selected directory to be the logical name 'FFP$INC'. The VMS command for this is:

```
$ DEF FFP$INC equivalence_name
```

2) Read the files on the distribution tape to the selected directory. In the example below, we use MTA0 as the tape drive and DUA0:[DA.VSP] as the directory in which the VSP code is to be installed. These should be changed as appropriate. The tape is supplied in VMS backup format.

Make sure there is not a write ring in the tape before mounting it on the drive. The VMS commands to read the tape are:

```
$ ALLOCATE MTA0:
$ MOUNT MTA0:/FOR/NOWRITE/DEN=1600
$ BACKUP /LOG/VERIFY MTA0: DUA0:[DA.VSP]
$ DISMOUNT MTA0:
$ DEALLOCATE MTA0:
```

A.1.3 Using the VSP Simulator and Parser to Generate Fully Linked Executable Images

3) Define the physical name of the simulator and compiler to be executable VMS programs. The VMS commands for this are as follows:

```
VSPS :== $FFP$INC:VSPS.EXE
PARSE :== $FFP$INC:PARSE.EXE
```

4) The 'DEF' in 1 above and the definitions of PARSE and VSPS may be system wide commands or be included in the LOGIN.COM file for anyone who will be using the VSP software.

5) A command file TEST.COM and a file READ.ME are contained in the tape. The file READ.ME contains a copy of these instructions and may contain additional information. The file TEST.COM serves to validate that all programs have been correctly installed. To validate the installation copy TEST.COM to the directory of a VSP user and execute the command file (@TEST). If all goes well, a file VALDTE.ERR will be created and it will end with the following message:

```
Test(0:1) INSTALL completed with no errors.
```

If you do not receive this message then there may have been some error in the installation procedure. Please go over the above procedure and recheck everything. If this does not clear up all problems please contact your ZORAN representative.

6) To use the VSP simulator interactively you need only type: 'VSPS'. Note that the simulator is set up to use a VT100 compatible terminal. If you have a different terminal, you must create a file 'TERMINAL.ID' in the directory in which you are running. This file should contain the single character '1'. In this dumb terminal mode you will retain full VSP simulator capability but menus will scroll on the screen and the prompt lines will not be emphasized as they are on VT100 compatible terminals.

7) The high resolution graphics routines require a DEC VT240 compatible graphics terminal. If you do not have such a terminal you should not select this menu option. High resolution graphics under VMS is available in versions 2.2 and higher. MAKE SURE YOUR TERMINAL PORT IS

SET FOR NOWRAP IF YOU ARE USING HIGH RESOLUTION GRAPHICS. THE COMMAND TO SET THIS IS:

```
SET TERM /NOWRAP
```

The remaining steps explain how to link your application with the VSP parser. Command files are provided to do this easily. Before we describe those we will describe how to do the linking manually. Understanding what the steps are will be helpful in interpreting error conditions that may arise.

8) Assume you have prepared a VSP program called 'SOURCE.FFP' and you wish to compile and execute it. (Note: the program suffix must be FFP and it must contain a 'main' subroutine called 'vspop'.) The procedure is as follows:

A) Convert your source program to 'C' code using the VSP compiler. The VMS command for this is:

```
$ PARSE SOURCE.FFP
```

As a part of the compilation process several files with the prefix 'SOURCE' will be created. These are:

'SOURCE.C' - main output to be processed with 'C' compiler.

'SOURCE.PH' - initialization include file containing VSP instructions.

'SOURCE.CK' - include file to check size of compile time initialized fields.

'SOURCE.SUB' - include file containing subroutines generated from instruction blocks.

If you wish to change any of the VSP compiler options you should include these in quotes on the line in which you invoke the VSP compiler. In the following example the options 'CLINE' and 'CCKSIZE' are set (this clears the options 'LINE' and 'CKSIZE' which are the compiler defaults.

```
$ PARSE SOURCE.FFP "-CCKSIZE" "-CLINE"
```

B) Next you must compile the above output with the VMS 'C' compiler. The command for this is:

```
$ CC SOURCE
```

Remember to have the statement '#include "FFP$INC:FFP.H"' in the file 'SOURCE.FFP' and to use 'SET DEF' to define FFP$INC to be the physical name of the directory for the VSP files. If you have any syntax errors the line numbers of the errors may refer to either the original file 'SOURCE.FFP' or to any of the generated files containing 'C' code listed above. In the latter case you may determine which lines in the original file generated the errors by looking for comments in the file with the error. Lines in the form:

```
/* 'SOURCE.FFP':nnn */
```

give the original file and line number (nnn) which give rise to the code that follows immediately.

In general errors in the 'C' code included in 'SOURCE.FFP' will be flagged with reference to that file. Errors in the 'C' code that is generated by the VSP compiler will be referenced relative to the files output by the compiler. The compiler should not generate invalid 'C' code but it does copy arithmetic expressions without parsing them itself and these may contain syntax errors.

C) The output from the 'C' compiler is a linkable module 'SOURCE.OBJ.' This must now be linked with the remainder of the emulator code which is in the library: 'FFP$INC:FFP.OLB.' The main program is not included in the library. Instead, a copy is provided as 'FFP$INC:EMAIN.OBJ.' The commands to create and executable module that includes 'SOURCE.OBJ' are :

```
$ DEF LNK$LIBRARY SYS$LIBRARY:VAXCRTL.OLB
$ LINK SOURCE,FFP$INC:EMAIN,FFP/LIB
```

The first line tells the linker to use the 'C' library files. The name for these files may change with different VMS versions and/or different simulator releases. The second line in the command file does the linking. This will create an executable file 'SOURCE.EXE'.

D) To begin execution type:

```
$ RUN SOURCE
```

To execute the code you generated in SOURCE.FFP choose option '8' in the *Main* Menu and subroutine 'vspop' will be called.

E) Command files are provided to simplify the above process. However, it is important that you understand what the files are doing as outlined above so you can interpret any error conditions that arise. There are two command files. LVSPLINK does a complete parse, compile and link for one to eight '.FFP' files. It cannot be used if you want to specify command line options to the parser. In that case you must do a manual parse and compile.

There is a command, @FFP$INC:INLARGE, to set the VSP definitions. It can be invoked from the VSP user's LOGIN.COM. However make sure that FFP$INC is defined before this command file is invoked.

This command file contains the following lines:

```
$ VSPS :== $FFP$INC:VSPS.EXE
$ PARSE :== $FFP$INC:PARSE.EXE
$ LVSP :== @FFP$INC:LVSP.COM
$ LVSPLINK :== @FFP$INC:LVSPLINK.COM
```

The first two lines define VSPS and PARSE. The next two lines define the command files for parse, compile and link and for link only. After this you can do a link as follows.

```
LVSPLINK FILE1 FILE2 ...
```

Do not include the '.OBJ' suffix on the command line. The executable file created will be 'FILE1.EXE'. Similarly you can parse, compile and link up to eight parser input files (with suffix '.FFP') using the command:

```
LVSP FILE1 FILE2 ...
```

Again do not include the '.FFP' suffix in the command line. The executable file created will be 'FILE1.EXE'.

A.1.4 Using Sharable Images Under VMS

When you link your application to the VSP simulator as described above you will create a large '.EXE' file that includes a copy of all the simulator code. If you wish to link many separate applications, this can require substantial disk space. In addition, because you are actually linking the entire simulator, the time to perform the link is fairly long. These space and time problems can be avoided by using sharable images under VMS. THUS, IT IS RECOMMENDED THAT YOU USE SHARABLE IMAGES WHENEVER POSSIBLE. THERE ARE TWO NOTES OF CAUTION. YOUR SYSTEM MUST BE INSTALLED TO SUPPORT SHARABLE IMAGES. IF YOU ARE USING A VERSION OF VMS OTHER THAN THE ONE ON WHICH THE SIMULATOR RELEASE WAS PREPARED, PROBLEMS ARE MORE LIKELY TO ARISE WITH SHARABLE IMAGES THAN WITH SIMPLE LINKING TO A LIBRARY.

A.1.5 Instructions for Installing the Zoran VSP Simulator as a Sharable Image

9) Make sure you have completed at least steps 1 and 2 for reading the distribution tape.

10) Execute the command file '@FFP$INC:INSHARE'. Note: this command file should be included in the 'LOGIN.COM' file of anyone using the simulator regularly. It must occur after the definition of FFP$INC. Its contents follow.

```
$ VSPS :== $FFP$INC:VSPS.EXE
$ PARSE :== $FFP$INC:PARSE.EXE
$ VSP :== @FFP$INC:VSP.COM
$ VSPLINK :== @FFP$INC:VSPLINK.COM
$ DEF VSP$SHARE FFP$INC:VSP16SHR.EXE
```

(NOTE: IF YOU WISH TO LINK USING BOTH SHARABLE IMAGES AND FULL LINKS OF COMPLETE EXECUTABLE IMAGES, USE THE COMMAND FILE: FFP$INC:INBOTH)

After executing this command you can link one to eight '.OBJ' files to the simulator by entering

```
VSPLINK FILE1 FILE2 ...
```

Do not include the '.OBJ' suffix on the command line. The executable file created will be 'FILE1.EXE'. Similarly you can parse, compile and link up to eight parser input files (with suffix '.FFP'). Using the command:

```
VSP FILE1 FILE2 ...
```

Again do not include the '.FFP' suffix in the command line. The executable file created will be 'FILE1.EXE'.

The definition of 'VSP$SHARE' is necessary to tell the linker where to access the sharable image to link with your object code.

11) To validate the installation using sharable images copy SHTEST.COM to the directory of a VSP user and execute the command file (@SHTEST). If all goes well, a file VALDTE.ERR will be created and it will end with the following message:

```
Test(0:1) SHARE_INSTALL completed with no errors.
```

If you do not receive this message then there may have been some error in the installation procedure. Please go over the above procedure and recheck everything. If this does not clear up all problems please contact your ZORAN representative.

A.1.6 Summary of Command Files

INBOTH.COM - Set up names for both sharable image and independent module linking.
Names defined: VSPS, PARSE, VSP, LVSPL, VSPLINK and LVSPLINK

INLARGE.COM - Set up names for linking independent modules.
Names defined: VSPS, PARSE, LVSP and LVSPLINK

INSHARE.COM - Set up names for sharable image linking.
Names defined: VSPS, PARSE, VSP and VSPLINK

SHTEST.COM - Sharable image installation test.

TEST.COM - Independent linking installation test.

A.1.7 Summary of VMS Simulator Logical Names

VSPS - Invoke VSP simulator.
Defined to be FFP$INC:VSPS

FFP$INC - Home directory of VSP simulator files.

LVSP - Link one to eight '.OBJ' files to simulator.
Defined to be @FFP$INC:LVSPL

LVSPLINK - Parse compile and link one to eight '.FFP' files.
Defined to be @FFP$INC:LVSPLINK

PARSE - Parse a '.FFP' file generating 'C' code as output.
Defined to be FFP$INC:PARSE.EXE

VSP - Link one to eight '.OBJ' files to simulator using sharable images.
Defined to be @FFP$INC:VSPL

VSPLINK - Parse compile and link one to eight '.FFP' files using sharable images.
Defined to be @FFP$INC:VSPLINK

A.1.8 VMS Versions for VSP Simulator Version 2.3-5

VMS Operating System: 4.2

'C' Compiler and Libraries: 2.1-007

A.2 VSPS Installation Under ULTRIX

The steps for installing the VSPS and language under ULTRIX are described below. In all cases it is assumed that you are using the 'C' shell and not the Bourne shell.

1. Set the shell variable "ffp_home" to the directory containing the VSP files. The ULTRIX command for this is:

   ```
   set ffp_home=physical_name
   ```

2. Read the files on the distribution tape to the selected directory. In the example below we use mt0 as the tape drive and /da/vsp as the directory where the VSP code is to be installed. These should be changed as appropriate. If a different tape drive is used, include it as part of the tar key below. See Volume 1 of the UNIX programmers manual under tar(1) for more details.

   The tape is supplied in tar format. The file names are relative and will be copied to the directory you are connected to when you run tar. **Make sure there is no write ring in the tape before mounting it on the drive.**

   The ULTRIX commands to read the tape are:

   ```
   cd /da/vsp
   tar xv ./vsp
   ```

3. Define the physical name of the simulator and compiler to be executable ULTRIX programs. The ULTRIX commands for this are as follows:

   ```
   alias parse directory_name/parse
   alias vsps directory_name/vsps
   ```

4. It is a good idea to include the "set" in 1 above and the aliasing of "parse" and "vsps" in the ".cshrc" file for all users of the VSP software.

5. A command file "test" and a file "read.me" are contained in the tape. The file "read.me" contains a copy of these instructions and may contain additional information. The file "test" validates correct program installation. To validate the installation, copy "test" to the directory of a VSP user and execute "test". If all goes well, a file "valdte.err" will be created and it will end with the following message:

   ```
   Test(0:1) INSTALL completed with no errors.
   ```

   If you don't get this message, there may have been an error in the installation procedure. Go over the above procedure and recheck everything. If this does not clear up all problems, contact your ZORAN representative.

6. To use the VSPS, type "VSPS". Note that the VSPS is set up to use a VT100 compatible terminal. If you have a different terminal, you must create a file 'TERMINAL.ID' in your directory, with the single character "1". Note that in ULTRIX the file name must be all upper case. In this dumb terminal mode you will have full VSPS capability, but menus will scroll on the screen instead of flashing, and the prompt lines will not be emphasized as they are on VT100 compatible terminals.

7. The high resolution graphics routines require a DEC VT240 compatible terminal. If you do not have such a terminal, do not select this menu option.

8. Assume you have prepared a VSP program called "source.ffp" and you want to compile and execute it. Note that the program suffix must be ".ffp.". The procedure follows:

   a. Convert your source program to 'C' code using the VSP compiler. The ULTRIX command for this is:

      ```
      parse source.ffp
      ```

   As a part of the compilation process several files with the prefix "source" will be created. These are:

   source.c - main output to be processed with the 'C' compiler
   source.ph - initialization include file containing VSP instructions
   source.ck - include file to check size of compile time for initialized fields
   source.sub - include file containing subroutines generated from instruction blocks

   If you want to change any of the VSP compiler options, include these in quotes on the line where you invoke the VSP compiler. In the following example the options "CLINE" and "CCKSIZE" are set; this clears the options "LINE" and "CKSIZE", the compiler defaults.

   ```
   parse source.ffp -CCKSIZE -CLINE
   ```

   b. Next, compile the above output with the ULTRIX 'C' compiler. The command for this is:

      ```
      cc source.c
      ```

   Remember to put '#include "directory_name/ffp.h"' in the file "source.ffp", where "directory_name" is the directory containing the VSP files. If there are syntax errors, the line numbers of the errors may refer to either the original file "source.ffp" or to any of the generated files containing the 'C' code listed above. In the latter case you may determine which lines in the original file generated the errors by looking for comments in the file with the error, in the form:

      ```
      /* "source.ffp":nnn */
      ```

This gives the original file and line number (nnn) which generated the code immediately following.

In general, errors in the 'C' code in "source.ffp" will be flagged with reference to that file. Errors in the 'C' code generated by the VSP compiler will be referenced relative to the files output by the compiler. The compiler should not generate invalid 'C' code, but it does copy arithmetic expressions without parsing them and these may contain syntax errors.

c. The output from the 'C' compiler is a linkable module "source.o". This must now be linked with the remainder of the Simulator code in the library, "ffp.a". The ULTRIX command to perform this link is:

```
f77 -lm source.o directory_name/ffp.a - 14014 -o exe_name
```

The f77 version of the linker command file must be used because of the FORTRAN code in "ffp.a". This will create an executable file "exe_name".

d. To begin execution, type:

```
exe_name
```

To execute the code you generated in "source.ffp" choose option '8' in the *Main* Menu.

A.3 VSPS Installation On MS-DOS

CONTENTS

1 TERMS

a) In this section 'DOS' refers to the Microsoft MSDOS operating system, not the IBM mainframe operating system called DOS. MSDOS is called PCDOS when running on the IBM personal computer family.

b) 'Computer' means an IBM PC/AT or compatible (see Required Hardware).

c) 'Simulator' and 'VSPS' refer to the VSP Simulator program provided by Zoran.

d) 'User simulator' refers to a simulator created by linking user VSP code.

e) 'simulator' (lower case) usually applies to both the VSPS and user simulators.

f) 'VSPE' refers to Zoran's Vector Signal Processor Evaluation board for the IBM PC/AT and compatibles.

g) Directories are 'siblings' if they share a common parent. Thus the directories \ZORAN\VM and \ZORAN\VEXAMPLE are siblings because their parent is \ZORAN.

h) DOS commands and filenames are CAPITALIZED for emphasis although DOS actually ignores case.

i) When referring to a product, the word 'version' may be abbreviated 'v', or omitted. Examples: 'VSPS v2.35', 'DOS 3.1'.

2 REQUIREMENTS

To install and run the Simulator you need the following:

2.1 Required Hardware

a) An IBM PC/AT computer or close compatible (example: Compaq Deskpro 286). Older PCs such as the PC/XT have an 8-bit expansion bus and will not accept future Zoran development boards. Also, the speed of an AT-class machine is needed for linking user simulators in a reasonable time.

b) A hard disk with at least two megabytes space available

c) 640K of RAM.

> NOTE: the Simulator requires 640K of RAM, less the amount for DOS. 'Resident' software loaded after DOS must not occupy more than about five Kbytes. The DOS command 'CHKDSK' will tell you how much disk and RAM space is available. CHKDSK displays a summary which includes the following lines:

```
xxxxxxx bytes available on disk  -- item 'b' above
 xxxxxx bytes free  -- should be >578000
```

2.2 Recommended Hardware

For computers other than Compaq:

* Color Graphics Adapter ("CGA") or Enhanced Graphics Adapter ("EGA") (or compatible boards) and appropriate monitor, on PCs.

The Simulator can use an IBM monochrome adapter (text-only display), but will display only 'character' plots of signals; a color adapter allows 'graphic' plots.

The present Simulator does not use color or the special features of the EGA, so the least-cost display for graphic plots would be the CGA and a "composite" green or amber monitor.

2.3 Optional Hardware

* An 80287 math co-processor.

The Simulator automatically uses a math co-processor if one is present. This increases the speed of floating-point ("real-number") operations.

2.4 The VSP Evaluation Board (VSPE)

With Zoran's Vector Signal Processor Evaluation (VSPE) board, you can test your VSP programs on an actual VSP as well as via simulation. The VSPE package includes all software for creating user simulators with VSPE support. This software installs automatically.

2.5 Software Required to Operate the Simulator

* MS/PCDOS Operating System, version 3.1. Version 3.2 supports the IBM Convertible and is ***not*** required. Version 3.00 should work, but is not tested by Zoran.

NOTE: The DOS 'VER' command displays the version number.

2.6 Software Required to Create User Simulators:

* Microsoft's 'C' Compiler, version 3.00
* Phoenix Software's Associates 'Plink86' Object Linker, version 1.47 or 1.48.

NOTE: The current Plink86 version is 1.48. The VSPS link files will work for 1.47 and 1.48, but we recommend that users of v1.47 contact Phoenix for an upgrade to 1.48 as support for 1.47 may be dropped in the next release of the VSPS.

The file OVERLAY.LIB which comes with Plink86 is specific to a given Plink86 version; when upgrading be sure to replace this file.

2.7 Software for User Simulators Using FORTRAN:

* Microsoft's FORTRAN compiler, version 3.31 or higher.

3 INSTALLATION

The Simulator software is distributed on a set of diskettes. An installation procedure on the first diskette will automatically copy the Simulator files to the proper directories on the hard disk.

This procedure will:

* expect to find a hard disk drive named C:
* use the DOS 'RESTORE.COM' program (see section 3.2: VSPS Automatic Installation Requirements, below)
* create a \ZORAN directory on drive C: if no such directory exists
* create several subdirectories under the \ZORAN directory
* create a \MSF directory on drive C: if no such directory exists

3.1 Getting Started Quickly

To use the Simulator immediately, just complete the following steps:

3.2. VSPS automatic installation requirements

3.3. VSPS automatic installation procedure

4.1. CONFIGURING THE PC: CONFIG.SYS and ANSI.SYS

4.2. CONFIGURING THE PC: TERMINAL.ID

...then proceed to section 7, "Running the Simulator".

To completely configure the Simulator, proceed to the section "Configuring the PC". Refer to the Checklist in section 9.

3.2 VSPS Automatic Installation Requirements

Two conditions must be true for the VSPS automatic installation procedure to work. (These do not apply to *VSPE* installation).

Requirements:

1) <u>'RESTORE.COM' must be on the DOS search path</u>

To see if this is true, enter the command 'PATH' at the DOS prompt on the hard disk. PATH will respond with either:

a) 'No path', or
b) 'PATH=', followed by a list of directories (the 'path directories')

For RESTORE.COM to be 'on the path',

* a PATH must be in effect (response 'b' above), and
* RESTORE.COM must be on one of the path directories

If both of these are true then RESTORE.COM is 'on the path', so proceed to "2) <u>No resident programs...</u>", below.

Otherwise:

* If no path is set ('a' above), set a temporary path by entering 'PATH \'. Verify that the path has been set by entering 'PATH' again. This time, PATH should respond

    'PATH=\'

* If RESTORE.COM is not on the hard disk, copy it from the DOS SYSTEM diskette to one of the path directories. (If you just entered 'PATH \', copy it to 'C:\', the root directory of the hard disk).

* If RESTORE.COM is already on the hard disk, copy it from its present directory to one of the path directories. (If you just entered 'PATH \', copy it to 'C:\', the root directory of the hard disk).

2) <u>No resident programs should be installed</u>
(example: Borland's 'SideKick')

SideKick is one resident program which interferes with RESTORE.COM. If your \AUTOEXEC.BAT file installs any resident programs, you should temporarily change your \AUTOEXEC.BAT file to disable their installation, reboot the PC, install the VSP Simulator, then undo the changes to \AUTOEXEC.BAT.

3.3 VSPS Automatic Installation Procedure

To install the Simulator files:

* Place release diskette #1 in drive A:

* Type the following command at the DOS prompt:

```
A:INSTALL
```

* Follow the instructions displayed by the installation program. Be sure to insert the diskettes in the proper order.

* When installation is complete, follow any supplemental instructions displayed by the installation procedure.

3.4 VSPE Automatic Installation Procedure

[Skip this section if you do not have a VSPE board]

To install the VSPE files:

* Place the release diskette in drive A:

* Type the following command at the DOS prompt:

```
A:INSTALL
```

* Follow the instructions displayed by the installation program.

* When installation is complete, follow any supplemental instructions displayed by the installation procedure.

4 CONFIGURING THE PC

4.1 CONFIG.SYS and ANSI.SYS

The file \CONFIG.SYS ('\' means 'on the root directory') should include the following statement as its first line to ensure correct operation of the Simulator:

```
DEVICE=ANSI.SYS
```

For the above statement to work, the file ANSI.SYS must also be on the root directory of the hard disk. ANSI.SYS is found on the DOS SYSTEM diskette.

Changes to \CONFIG.SYS do not take effect until you reboot your PC (hold down the CTRL and ALT keys, then press DEL).

4.2 TERMINAL.ID, Simulator Display Configuration File

If your system does not have a graphics display (e.g. a 'Color Graphics' or 'Enhanced Graphics' adapter with appropriate monitor), rename the file TERMINAL.IDX on the \ZORAN\VM\ directory to TERMINAL.ID. This disables 'graphic' plotting, which is not possible on a monochrome text adapter; use 'character' plotting instead.

4.3 Microsoft 'C' Compiler

The Microsoft 'C' Compiler should be installed and be accessible via the DOS search path. In addition, a 'SET INCLUDE=' command must be issued to tell the compiler where to find 'include' (header) files. Details of both operations appear in the section 'AUTOEXEC.BAT...' below.

4.4 Microsoft FORTRAN Compiler

The automatic installation procedure copies two Microsoft FORTRAN library files to the \MSF directory, creating the directory if necessary. If you do not wish to create user simulators using FORTRAN source code, the compiler itself is not required.

If you wish to create user simulators using FORTRAN, the Microsoft FORTRAN compiler should be installed on the directory listed in the file \ZORAN\VM\E.LNK (which currently specifies the \MSF directory).

4.5 PLink86 Linker

The PLink86 linker should be installed and be accessible via the DOS search path. In addition, a 'SET OBJ=' command must be issued to tell PLINK86 where to find its overlay library, OVERLAY.LIB. Details of both operations appear in the section 'AUTOEXEC.BAT...', below.

4.6 Printing Graphic Screens

The contents of the screen of a PC can be copied to the printer by pressing SHIFT-PrtSc (hold down SHIFT while pressing the */PrtSc key). This is knows as a 'screen dump'.

However, to obtain screen dumps of graphic screens (rather than text mode) you must:

* have a graphics display (of course);
* have a graphics printer such as an EPSON MX- or FX- series;
* run the MSDOS 'resident' command GRAPHICS.COM before running the Simulator. This may be done automatically when your PC is booted via the AUTOEXEC.BAT file (see the following section).

4.7 AUTOEXEC.BAT and AUTOEXEC.TXT

For the VSPS and its utilities to run from directories other than \ZORAN\VM, the DOS search path must be set properly.

Also, for Microsoft 'C' and the PLink86 linker to work, the DOS search path must be set properly and two SET commands must be issued. The usual place for these things to be done is in the \AUTOEXEC.BAT file, which is executed when the PC is booted.

> NOTE: spaces are significant in the SET command. *There should not be a space before the equal sign ('=').*

Lastly, a program called GRAPHICS must be run to enable graphic screen dumps (see previous section).

The following 'fragment' of an AUTOEXEC.BAT file, stored as \ZORAN\VM\AUTOEXEC.TXT, performs all the required operations. Your hard disk probably has an existing \AUTOEXEC.BAT file; *do not simply replace it with the contents of AUTOEXEC.TXT*. Instead, use AUTOEXEC.TXT as a guide when changing your \AUTOEXEC.BAT, with appropriate changes if your directory layout differs. Lines beginning with ':' are comments.

```
: AUTOEXEC.TXT: Fragment of \AUTOEXEC.BAT for VSPS
: Last edit 8/1/86 JJC
:
: Set DOS prompt to show the search path
PROMPT $p$g $a
:
: Set DOS search path so DOS can find 'C' compiler
: and PLink86. Assumes 'C' compiler is on \MSC,
: FORTRAN compiler (if installed) is on \MSF, and
: PLINK86.EXE is on \UTILS.
: Edit this string into PATH command in \AUTOEXEC.BAT
PATH \UTILS;\ZORAN\VM;\MSC;\MSF;
:
: Set 'INCLUDE' environment variable so Microsoft 'C'
: can find header files for itself and the VSPS.
SET INCLUDE=\MSC\INC;\ZORAN\VM
:
: Set the 'OBJ' environment variable so PLink86
: can find the file OVERLAY.LIB (assumed here to be
: on the \UTILS directory). \ZORAN\VM\E.LNK tells
: where to find other library and object files.
: NOTE: NO SPACES BETWEEN 'OBJ' and '='.
SET OBJ=\UTILS
:
: Install the DOS graphic print-screen facility.
: To be done after all PROMPT, PATH and SET commands.
: (GRAPHICS need not be installed on text-only systems)
: GRAPHICS.COM must be on the root directory or
: on the PATH.
GRAPHICS
```

5 DIRECTORIES AND FILES

5.1 Simulator Directories and Files

Directory (directory description)

File	Description
\MSF (Microsoft FORTRAN files)	
MATH.LIB	Microsoft FORTRAN libraries;
FORTRAN.LIB	used to link user simulators
\ZORAN\VEXAMPLE (Examples for creating user simulators)	
EXAMPLE1.FFP	VSP source code example: Load, FFT, Store
DEFER1.FFP	Load, FFT, Store, with DEFER
EXAMPLE2.FFP	FFT of cosine, find max freq
EXAMPLE3.FFP	Overlapped FFTs (separate RAM sections)
EXAMPLE4.FFP	Use FORTRAN subroutines to create squarewave, print FFT and power spectrum
FORPACK.FOR	FORTRAN code for subroutines in EXAMPLE4
EXAMPLE4.LNK	"Link file" to link EXAMPLE4.OBJ and FORPACK.OBJ into a user simulator
TEST2.*	User simulator creation test
TESTVSPE.*	User simulator creation test with VSPE board support; installed with VSPE
\ZORAN\VM ('VM' = 'VSPS MAIN' directory)	
AUTOEXEC.TXT	Fragment of \AUTOEXEC.BAT
B.LNK	Beginning of user simulator link
CREATE.BAT	Parses, compiles, and links VSP source into a user simulator
CLEANUP.BAT	Deletes parse, compile, link results; leaves .FFP and .EXE
E.LNK	End of user simulator link
INSTALL1.BAT	From installation; can be deleted
PARSER.EXE	("Parser"); produces 'C' code from VSP source ('.FFP'). Driven by VP.BAT
README	(this file)
TEST1.*	Installation test (must run in this directory)
TRAILER	Message from installation; can be deleted
VP.BAT	Parses .FFP into .'C' (drives PARSER.EXE)
VC.BAT	Compiles .'C' into .OBJ (drives Microsoft 'C' compiler)
VL.BAT	Links .OBJ into a simulator (drives PLink86 linker)
VSPS.EXE	The VSP Simulator program
(others)	Used by installation test
\ZORAN\VSPUTILS (VSPS helpfiles and miscellaneous)	
(several)	Used in simulator execution

\ZORAN\VSPE (VSPE board support files, if installed)

*.OBJ	Object files for creating user simulators with VSPE support
*.HEX	VSP files for board test
INSTALL1.BAT	From installation; can be deleted
VSPE_B.LNK	Beginning of link with VSPE support
VSPE_E.LNK	End of link with VSPE support
VSPTEST.EXE	Board test (runs only in this directory)
LOADTEST.DAT	Used by board test

\ZORAN\VUSER (Your source files go here)

NULLFILE	Used for installation; can be deleted

NOTE: You can make other VSP user directories, but they must have \ZORAN as their parent directory. Choose any names other than the directory names listed above (\ZORAN\VUSER can be renamed).

5.2 Directories and Files for Creating User Simulators

Files	Suggested	If elsewhere, change
Microsoft 'C' ("MSC")	\MSC	'PATH' in \AUTOEXEC.BAT
MSC '.H' files	\MSC\INC	'SET INCLUDE' in \AUTOEXEC.BAT
MSC '.LIB' files	\MSC\LIB	
PLink86 Linker	\UTILS	'PATH' in \AUTOEXEC.BAT
OVERLAY.LIB	\UTILS	'SET OBJ' in \AUTOEXEC.BAT
MATH.LIB	\MSF	\ZORAN\VM\E.LNK
FORTRAN.LIB	\MSF	\ZORAN\VM\E.LNK

(The rest of the FORTRAN compiler files are only required if you develop VSP applications using the FORTRAN language.)

Other FORTRAN compiler files	\MSF	'PATH' in \AUTOEXEC.BAT

5.3 Filename Extensions

The following DOS filename extensions are used by the VSP Simulator package:

<u>Extension</u>	<u>Used for</u>
.BAT	DOS batch (command) file
.BIN	VSP object code (binary)
.C	'C' source (or Parser output)
.CK	Intermediate file produced by Parser
.DAT	Installation test data
.ERR	'C' or FORTRAN compilation error file
.EXE	MS-DOS executable program
.FFP	VSP source code
.FOR	FORTRAN source
.GEN	Installation test data
.H	'C' header ("#include") file
.IEE	Installation test data
.INP	Installation test input (keystrokes)
.LER	Link error file
.LNK	Link instructions for PLink86 linker
.MAP	Link map of user simulator
.MSG	Simulator help files
.OBJ	Object code ('C' or FORTRAN compiler
.PH	Intermediate file produced by Parser
.SPS	Installation test data
.SUB	Intermediate file produced by Parser
.TST	Installation test commands
.TXT	Text files (human-readable)
.(none)	Macro files, or installation text files

6 TESTS

6.1 VSPS Installation Test

The Installation Test verifies proper installation of the Simulator files. <u>This test is performed automatically at the end of the automatic installation procedure.</u>

To run this test, go to the \ZORAN\VM directory and enter:

```
TEST1
```

This test takes several minutes to complete. Expect a message with the words "completed with no errors".

6.2 User Simulator Creation Test

The User Simulator creation test checks that the Parser, Microsoft 'C' Compiler, Microsoft FORTRAN library files, and the PLink86 Linker are in place.

To run this test, go to the \ZORAN\VEXAMPLE directory and enter:

```
CREATE EXAMPLE1
```

This creates a user simulator called EXAMPLE1.EXE, which can then be run by entering:

```
EXAMPLE1
```

This procedure, CREATE, is the same one you will use when creating your own User Simulator.

You can test this user simulator via the command:

```
TEST2
```

A user simulator takes up substantial space on the disk. When satisfied with the results of TEST2, enter the following commands to delete EXAMPLE1.EXE and several other files which were generated by CREATE:

```
CLEANUP EXAMPLE1

DEL EXAMPLE1.EXE
```

6.3 VSPE Installation Test

(This section applies only if you have a VSPE board installed)

The VSPE Installation Test verifies proper operation of the VSPE board. This test is performed automatically at the end of the automatic installation procedure for the VSPE software.

To run this test, go to the \ZORAN\VSPE directory and enter:

```
VSPTEST
```

This test takes several minutes to complete.

6.4 User Simulator Test with VSPE support

This test checks the operation of the VSPE board with a user simulator, and is similar to the test described in section 6.2.

On the \ZORAN\VEXAMPLE directory, create a user simulator with VSPE support from the VSP source file EXAMPLE1.FFP:

```
CREATE EXAMPLE1 VSPE
```

To start the test, enter:

```
TESTVSPE
```

7 RUNNING THE SIMULATOR

To run the Simulator, go to the \ZORAN\VM directory and enter:

```
VSPS
```

You can also run the Simulator or its command files from any sibling directory of \ZORAN\VM. (For this to work, the DOS PATH must be set properly. See section 3, "Configuring the PC").

You will typically use the directory \ZORAN\VUSER when developing your own ("user") simulator from VSP source code. During the Installation Test the Simulator expects the current directory to be \ZORAN\VM.

> NOTE: The current release of the Simulator cannot be run from any directory whose parent is not \ZORAN, since it expects certain of its files to be located in \ZORAN\VSPUTILS, which is a sibling of \ZORAN\VM.

8 DEVELOPING USER SIMULATORS

To develop your own VSP code using the simulator facilities, you create a 'user simulator' by parsing and compiling your code, then linking the resulting object code with the rest of the simulator. The directory \ZORAN\VUSER is provided for user simulator files, but you may rename this directory or create one or more sibling directories for your VSP code (e.g. \ZORAN\MYPROJ1, \ZORAN\MYPROJ2, etc).

NOTE: In the present VSPS release, user simulators will not operate on any directory that is not a sibling of \ZORAN\VSPUTILS.

8.1 Single User Module (Simplest Case)

The simplest way to create a user simulator is by putting all VSP and 'C' source code in a single module:

a) Write VSP source code, possibly intermixed with 'C' statements, using your favorite text editor.

This source file must have the '.FFP' extension and must reside on a directory whose parent is \ZORAN. (Here we use 'EXAMPLE1.FFP', a file found on \ZORAN\VEXAMPLE)

b) Parse the VSP source (e.g. EXAMPLE1.FFP) into a 'C' source file (e.g. EXAMPLE1.C), using the VP command. Omit the '.FFP' extension:

```
VP EXAMPLE1
```

c) Compile the 'C' source code created by the parse operation (e.g. EXAMPLE1.C), using the VC command. Omit the '.C' extension:

```
VC EXAMPLE1
```

d) Link the resulting object code (e.g. EXAMPLE1.OBJ) to create a user simulator (e.g. EXAMPLE1.EXE). Omit the '.OBJ' extension:

```
VL EXAMPLE1
```

As an alternative to 'b' through 'd' you can VP, VC and VL in one operation by entering:

```
CREATE EXAMPLE1
```

e) Run your user simulator by entering its name, e.g.:

```
EXAMPLE1
```

and invoke the 'vspop' option from the *Main* Menu to test your application code using the full power of the VSPS facilities. The 'vspop' function in your .FFP module is the entry point to your application.

8.2 Single user module with VSPE support

If the VSPE board and support software have been installed, you can include VSPE board support in your user simulator via the second form of the 'CREATE' and 'VL' commands.

Following the example of the previous section:

```
CREATE EXAMPLE1 VSPE
```

or

```
VL EXAMPLE1 VSPE
```

Run your user simulator by entering its name, e.g.:

```
EXAMPLE1
```

The 'VSP Evaluation Board Control' option of the main menu leads to a submenu which enables you to control the VSPE board and invoke your code ('vspop' function). As before, the 'vspop' function in your .FFP module is the entry point to your application.

8.3 Multiple User Modules

Creating a user simulator from multiple user modules is similar to the single-module case, with several additions. Modules can be VSP source (.FFP), "C" (.C), or FORTRAN (.FOR). The files EXAMPLE4.FFP, FORPACK.FOR, and EXAMPLE4.LNK on \ZORAN\VEXAMPLE form a multi-module example.

Follow these steps:

a) Only one .FFP module (here, EXAMPLE4.FFP) provides a 'vspop' function.

b) Parse all .FFP sources, then compile 'C' sources (VC...) and FORTRAN sources (VF...) into object files. In our example we would do the following:

```
VP EXAMPLE4

VC EXAMPLE4

VF FORPACK
```

c) Provide a "link file" -- a short text file which tells the linker how to link your multiple-module simulator. The link file should:

* have the same name as the file which defines 'vspop',
* have the extension'.LNK',
* must consist exclusively of one or more lines of the form:

```
FILE name1 name2 name3 ...
```

where the 'names' are the names of you compiled object modules. Each line is terminated by carriage-return/linefeed. PLink assumes 'name' means 'name.OBJ', so omit the '.OBJ' suffix unless you use a different one.

In our example, the link file EXAMPLE4.LNK contains the following line:

```
FILE example4 forpack
```

d) Link the user simulator as you would a single user module. For example:

```
VL EXAMPLE4
```

Again, we could have performed only the following two steps:

```
VF FORPACK

CREATE EXAMPLE4]
```

CREATE and VL automatically perform a multiple-module link if they find a .LNK file of the proper name. The link file is combined with \ZORAN\VM\B.LNK and \ZORAN\VM\E.LNK, which PLink86 uses to link the user simulator.

If no link file is found, CREATE and VL perform a single-module link.

8.4 Multiple user modules with VSPE support

If the VSPE board and support software have been installed, you can include VSPE board support in your user simulator via the second form of the 'CREATE' and 'VL' commands just as with single user modules. Following the example of the previous section:

```
CREATE EXAMPLE4 VSPE
```

or

```
VL EXAMPLE4 VSPE
```

9 CHECKLIST

a. Requirements (section 2)

* PC/AT or close compatible, and DOS 3.1	( )
* At least two megabytes available on hard disk	( )
* 640K RAM	( )
* No resident programs over 5K long	( )

b. Getting started quickly (assumes 'Requirements' are met)

* Automatic installation (A:INSTALL) performed:	( )
* If graphics display, TERMINAL.ID deleted	
from \ZORAN\VM directory	( )
* CONFIG.SYS includes 'DEVICE=ANSI.SYS'	( )
* ANSI.SYS file is on PC's root directory	( )
* PC has been rebooted so changes to CONFIG.SYS	( )
and AUTOEXEC.BAT take effect	

c. For developing user simulators (assumes 'a' and 'b')

* AUTOEXEC.BAT: PATH includes	
\ZORAN\VM	( )
Microsoft 'C' directory	( )
PLink86 directory	( )
* Microsoft 'C' compiler installed	( )
* AUTOEXEC.BAT: SET INCLUDE specifies	
Microsoft 'C' header (*.H) file directory	( )
\ZORAN\VM	( )
* AUTOEXEC.BAT: SET OBJ specifies	( )
location of PLink86 'OVERLAY.LIB' file	
* PC has been rebooted so changes to CONFIG.SYS	( )
and AUTOEXEC.BAT take effect	

d. For FORTRAN compilation (optional)

* Microsoft FORTRAN compiler installed	( )
* AUTOEXEC.BAT: PATH includes	
Microsoft FORTRAN directory	( )
* PC has been rebooted so changes to CONFIG.SYS	( )
and AUTOEXEC.BAT take effect	

e. For creating user simulators with VSPE support

* VSPE Automatic installation (A:INSTALL) performed:	( )

APPENDIX B

SIMULATOR DIFFERENCES BETWEEN THE VAX AND PC

The versions of the VSP simulator supported on the VAX and the PC are identical in structure, execution and usage. However, the larger memory address space of the VAX allows a higher level of functionality in certain areas of simulator operation than does the version provided on the PC. The differences between the two simulators are tabularized below:

Specification	MS-DOS version	VAX versions
VSP program and data	16K words total	64K words each
Maximum IEEE FFT size	1K points	8K points
Number of simulated VSPs	2	8
Application Library	Not in this release	Included

APPENDIX C

ENOUGH 'C' TO ALLOW PROGRAMMING IN FORTRAN

C.1 Overview

The intent of this appendix is to provide a simple overview of the 'C' programming language. It is not intended to cover the language in its entirety. However, having read this section, it is likely that the examples provided in the manual will be more understandable for those not familiar at all with 'C'. For a more complete treatment of the language, please see reference 4 in Appendix E or any other suitable 'C' reference manual.

C.2 Declaring Variables in 'C'

When writing subroutines that are to be called from 'C' in another language, it is necessary to know how variables are declared in 'C'. There are primarily four variable types with which to be concerned. They are: *integers* (32-bit twos complement), *short integers* (16-bit twos complement), *floating-point* (32-bit floating-point) and *double-precision floating-point* (64-bit floating-point). In addition, each of the integer types can be declared to be unsigned.

All variables in 'C' must be declared before they are used. The declaration may be either prior to and external to a subroutine, or just after the bracket "{" in the subroutine body. In the former case the variables are external and available to all subroutines, like COMMON variables in Fortran. In the latter case they are defined only within the body of the subroutine. The end of all statements in 'C' are delimited with a semicolon.

Declarations of the four variable types are illustrated below.

```
int a, b, ix;           /* Declares three 32-bit integer variables with names a,
                           b and ix */
short int kshort;       /* Declares a single 16-bit integer named kshort */
unsigned int uns;       /* Declares a 32-bit unsigned integer named uns */
float ireal, imag;      /* Declares two single-precision floating-point values
                           named ireal and imag */
double test;            /* Declares a double precision value named test */
```

C.3 Declaring Arrays in 'C'

Within the VSPS, host memory is modeled as an array of 16-bit integers. The real and imaginary parts are stored in even and odd addresses, respectively. Data may be read from, and written to, this array in order to provide input and output from the simulated VSP. (Technically it is not an array, but part of a structure and is accessed by a pointer to a structure. However, by including the define for EXTRAM as discussed in Chapter, IX it can be accessed as an array.)

'C' denotes array subscripts using square brackets "[ ]". For multi-dimensional arrays, a series of square brackets is used. Arrays are declared similarly to variables as shown below.

```
int ary[20];                /* This declares an array named ary of 20 integers */
short int sary[10][15];     /* This declares a two-dimensional array named sary of
                               150 elements */
```

In 'C', all array subscripts begin at 0. The elements in *ary* as declared above are: *ary*[0], *ary*[1], ... , *ary*[19].

C.4 DO Loops and Assignment Statements in 'C'

This section will describe how to set up simple loops and assignment statements in 'C'.

The basic form of a loop in 'C' is illustrated below:

```
for (i=1; i<=100; i=i+1)
        {
                /*body of loop*/
        }
```

This is equivalent to the FORTRAN commands:

```
DO 50 I=1,100,1
C BODY OF LOOP

50 CONTINUE
```

The "i=1" and "i=i+1" parts of the "for" statement can be any valid 'C' statement. The statement "i=i+1" can be abbreviated as "i++", which indicates that "i" should be post-incremented (after it is used). The scope of the loop in 'C' is delimited with braces "{}" rather than with statement label numbers. The characters "<=" denote the relationship less than or equal to. The following table gives the 'C' equivalent for various FORTRAN relational operators:

FORTRAN	**'C'**	
.LT.	<	less than
.GT.	>	greater than
.LE.	<=	less than or equal to
.GE.	>=	greater than or equal to
.EQ.	==	logical equality (be sure to use a double equal sign; a single equal sign has a different meaning in 'C'
.NE.	!=	not equal
.AND.	&&	logical AND (warning as above)
.OR.	||	logical OR (warning as above)
.NOT.	!	logical negation

'C' allows the nesting of loops, as shown below:

```
for (i=1; i<101; i++)
   {for (j=3; j<17; j=j+2)
           {

                /* body of loop*/

           }/* end loop on j*/
   }/* end loop on i*/
```

This is equivalent to the FORTRAN code below.

```
        DO 10 I=1,100
        DO 20 J=3,16,2

C       BODY OF LOOP

20      CONTINUE
10      CONTINUE
```

A number of examples of 'C' assignment statements have been shown here as well as the examples in Chapter IX. The equal sign "=" is used to equate a variable with a value or result of an arithmetic operation. In this respect, 'C' maintains compatibility with Fortran.

There is a difference between 'C' and Fortran in the use of statements. In 'C', statements are separated with semicolons. It is perfectly acceptable to put multiple statements on a single line if they are separated by a semicolon. In Fortran, statements are separated by lines; it is not possible to have multiple statements on a given line. This construct is illustrated by referring to the "for" statement in the 'C' example above. Notice that the "for" statement is really composed of three assignment statements.

A number of differences exist between 'C' and Fortran expressions. One general difference is that all logical expressions using the operators in the previous table can be included in arithmetic expressions in 'C' because they are treated as integers. The value of a logical expression is integer 1 if the expression is true; otherwise it is 0. Similarly, arithmetic expressions can be used as logical expressions with 0 denoting false and any non-zero value being true. The arithmetic operators are the same in both languages.

APPENDIX D

FORMAL VSP PROGRAMMING LANGUAGE SPECIFICATION

D.1 Overview

Chapters IX and X in the manual present a solid foundation for the formal specification of the VSP programming language which is included in this appendix. The VSPS also has a "batch verification" mode which is useful for performing automated tests. Batch verification is covered in this appendix.

D.2 VSP Language Syntax

The VSP language is specified in a Backus-Naur Form (BNF) type of notation. This notation uses symbols as a shorthand way of describing the language construction:

Symbol	Meaning
;	indicates end of a statement
:	read literally as "is defined as"
|	read literally as "or"
-	used to tie words together - indicates a string of contiguous letters
"symbol"	used to define a symbol that is part of the syntax of a command

If you work with yacc on the UNIX system, you will recognize this as the same notation.

D.3 VSP Program

```
program: program_part PRGEND;

program_part:
program_part ffp_statement|
program_part VSPEND VSPBEG|
program_part VSPEND STRTEX VSPBEG|
program_part VSPEND STRTEX|
program_part VSPEND|
marker VSPBEG ;
```

VSPBEG and VSPEND are the delimiters "/#" and "#/" respectively. Everything outside of these is ignored by the VSP language, and written unchanged to the output file by the parser. PRGEND is the marker "##" that must always be at the end of the program.

D.4 VSP Program Markers

marker: ENDDEC | STRTEX | ENDDEC STRTEX | error | ;

The markers show the VSPS parser where the external declarations end in the 'C' source, and where the executable statements begin. The former is always required. The latter is necessary only if you have requested run time checks on the size of the instruction parameters.

ENDDEC is the marker "/#D#/" where external declarations end. At this point, write an #include statement to load the predefined array of partially built VSP instructions.

STRTEX is the marker "/#E#/" where executable code starts. At this point a subroutine is called to check the size of parameters defined at load time (not at VSP parse time).

The token "error" occurring in the syntax is to allow the parser to recover from syntax errors. If no correct input is found, the token "error" can match any sequence of inputs.

D.5 VSP Statements

```
ffp_instruction: label "::" ulffp_instruction |ulffp_instruction;

ffp_statement: ffp_instruction | instruction_set;

ulffp_instruction: elt_list "@" expression ";" |
        elt_list ";" |
        pseudo_op ";" |
        error ";" |
        ";" ;
```

The VSP language has three types of statements: VSP instructions, pseudo-operations (commands to the compiler and run time interpreter), and instruction blocks. With the latter you can load instruction parameters efficiently outside of loops. Instructions can be labeled so you can WAIT until a specified instruction has executed. In a multi-VSP system, instructions can be sent to a designated VSP by following the instruction with "@ expression". The expression is evaluated to determine the VSP number.

D.5.1 VSP Instructions

```
elt_list:elt_list "," name_elt |
iname name_elt | iname;

name_elt: tname ":" expression|
tname "=" expression;
```

The format of VSP instructions is the same as it is in the interactive mode. For example:

```
FFT NMBT : 64, RS=0, ADF=3, AS=0, R:0, FSIZ:64, EI=0, FPS:32,
        LPS:1, I:0, RBA:0;
```

Parameters can be set with LITERAL values or LOGICAL values. The designators show whether a parameter value is logical ":" or literal "=". Literal values are the exact value of the expression. Logical parameters use a more natural representation of the value for some parameters. You are given the translation of logical to literal values with the description of the instructions and in the instruction tutorial in the VSPS.

D.5.2 Pseudo-Operations

Pseudo-operations are commands to the VSP parser. They give the parser information (STATIC and DEFAULT), control VSP/host coordination (WAIT) and the disposition of VSP instructions (DEFER and IMMEDIATE), or control compiler options (OPTION). The formal syntax of the VSP pseudo-operations are presented below.

```
pseudo_op : STATIC sname_list | DEFAULT xelt_list|
WAIT wait_tail | DEFER expression | IMMEDIATE|
      OPTION option_list;

option_list : option_list "," set_option | set_option;

set_option : oname "=" expression | oname;

sname_list : sname_list "," sname | sname;

xelt : xtname ":" PNUMBER | xtname "=" PNUMBER;

xelt_list : xelt_list "," xelt | xelt |
      xelt_list error "," xelt;

xtname : iname "." tname | tname;

wait_tail : DONE | wlabel "@" expression | wlabel;

wlabel:wname":"expression| wname;
```

STATIC. This informs the compiler that certain variables are defined at compile or load time, and can be initialized. If a parameter is set to an expression with a variable not static, that parameter will be initialized at execution time.

DEFAULT. This gives the compiler default values for instruction parameters. You can set a default value for every occurrence of a parameter, or set it for a specific instruction. DEFAULT NMPT=5; will set the value of NMPT in every instruction using this parameter. DEFAULT LD.NMPT=5; will set the default value of NMPT only in the LD instruction. You can set the values of parameters with either literal values, using "=", or logical values, using ":".

WAIT. This causes 'C' code execution to be suspended until the specified VSP instruction is completed. DONE suspends execution until all issued VSP instructions have been executed. Specifying a label effects a WAIT for the labeled instruction to terminate. Following a label with a ":" and an expression with the value N suspends execution until the Nth instance of the labeled instruction has been executed. For example: WAIT STINS:4 @2; will effect a WAIT in VSP 2 until the instruction labeled STINS has been executed four times.

DEFER and **IMMEDIATE**. These are low-level operations for use with the JMPI instruction. DEFER causes subsequent instructions to be written to VSP external memory at the address specified in the "expression". IMMEDIATE turns off DEFER and causes subsequent instructions to be executed immediately.

Instructions which have been DEFERred cannot be controlled by the WAIT statement. If you use the DEFER and IMMEDIATE options you must handle coordination between the VSP and host directly, or rely on the "WAIT DONE;" statement.

When this program is executed, all VSP code except the JMPI instruction will be written to external memory. JMPI will then be executed and will start execution of the code in external memory. Execution from memory will cease when HLT is executed.

OPTION. This statement sets compiler options. SET and CLEAR (non VALUE) options can also be set at compile time, as explained above in the section on Executing the VSP Compiler. The compiler options are all shown in section 9.8.3.

D.5.3 Instruction Blocks

The syntax for instruction blocks is similar to that for 'C' subroutines. You can use VSPEND and VSPBEG inside instruction blocks; 'C' code placed between them is included in the subroutine created for the block. Each instruction block defines a 'C' subroutine of the same name and with the same set of parameters (all of type "int") as the instruction block name and parameters. A formal description of the instruction block language is given below.

```
instruction_set:instruction_set_head instruction_block;

instruction_set_head:build_begin"("parameter_list")";

build_begin:BUILD bname;

parameter_list:parameter_list","parm_name|parm_name|;

instruction_block:"{"instruction_list"}";
```

```
        instruction_list:instruction_list ffp_instruction|
        instruction_list VSPSND VSPBEG | ffp_instruction |
              VSPSND VSPBEG ;
```

Code for an instruction block is emitted at two distinct places. The first is within the subroutine, and the second is at the point in the calling program where the block is defined. The subroutine definition is written to a separate file to be read with "#include" where the "/#D#/" marker occurs. All fields that do not contain parameters for the block will be filled in when the block definition is executed. The remaining parameters will be filled in when the 'C' subroutine with the same name as the block is called.

An example of the use of the instruction block is given in section 9.8.6.

D.6 VSP Expressions and Names

```
        expression: expression PAREXP | expression.name |
               PNUMBER | PAREXP | ename;

        iname: ename: tname: wname: sname:
label: oname: parm_name: bname: NAME;
```

VSP expressions are not parsed by the VSP compiler. You may get a syntax error in the 'C' compiler from expression code copied by the parser. To trace this to your original source code, check the line number in the VSP source code comment.

The VSP language contains many different tokens for NAME. This allows executing different semantic actions based on context. All names begin with an alphabetic character. Following characters may be alphabetic, numeric or the underscore character "_". Upper and lower case characters are different. Name length should not exceed 20 characters.

D.7 Running The VSPS in Batch Verification Mode

You can run your program in a background mode, where the VSPS uses command files and can generate or check results for a program run. Select *Main* Menu option M-9. This mode allows large test cases or batch jobs to be built up in macro fashion from a series of files. After you select M-9, you are prompted to enter the name of the file containing the commands. The commands are described in detail below.

When the VSPS is run in this mode, it creates a file named VALDTE.ERR which lists the results of the tests. It will list the errors from the COMPAR command listed below, or will tell you that there were no comparison errors.

D.7.1 Batch Verification Mode Commands

Commands in this first set control execution and output format.

NAME name - sets name of test. When the command is executed, all notes in VALDTE.ERR will use "name" to identify output from this particular test. This command can be executed many times in one file.

BRANCH filename - tells the VSPS to branch to a new file and begin executing commands there. The VSPS returns to the original file when the BRANCHED file has completed execution.

VISIBLE - shows dialogue between the VSPS and file input.

INVISIBLE - disables VISIBLE if command was given.

MSGLEV LEVEL - sets message level. The variable "level" (declared int) is set to a number representing a pattern in the four least significant bits. The action of the bits is:

Bit	Action
0	(lsb) Enable requests for input
1	Turn on all output for instructions as executed
2	List instructions as executed
3	Enable display of timing information

For example, the command "MSGLEV 2" will activate printout just as will option '2' on Menu M-4 in the interactive mode.

SEED seed - sets the random number generator seed.

MOVE base1 base2 size - moves data of length size in external RAM location base1 to external RAM location base2.

SPS DES_FILE - the VSPS executes the simulator signal processing option from DES_FILE. This is the same as calling Menu M-5, only now the VSPS obtains the algorithm and its parameters from DES_FILE instead of interactively.

IEEE DES_FILE - the VSPS executes its IEEE signal processing options on DES_FILE. This is the same sort of command as SPS above, and is the same as calling Menu M-6. DES_FILE contains the commands you would enter interactively.

APP DES_FILE - the VSPS executes its applications library from DES_FILE. The procedure is the same as that for SPS above, but it is called from Menu M-7, and DES_FILE contains the command sequence.

FFPOP - tells the VSPS to execute the code in ffpop. Your program should not contain any prompts to the user. If you have to pick up parameters in your program, open a file just as you would in 'C', and get the parameters from this file.

The commands in this next set control the VSPS software operating system.

QOPT n - sets the VSP/host coordination type. Exactly like setting options 1,3,4 and 7 in Menu M-4-21, except that n ranges from 0 to 3 (0 represents option 1, 1 represents option 3, 2 represents option 4, and 3 represents option 7).

ALLQ - tells the VSPS to model all fetch software and hardware queues. This command does the same thing as option 9 in Menu M-4-11.

HWQO - tells the VSPS to model only the hardware queue. This command is the same as option 2 in Menu M-4-22, except that only the hardware queue is enabled. If you want to change the VSP/host interaction model from this default, use QOPT.

EXDMEM - causes the VSPS to do an instruction fetch from the program memory addressed by the freeze pin of the VSP. This option moves the fetch queue of the VSPS software operating system into program memory, and enables you to access program memory if you do your own instruction fetching (using the DEFER pseudo-operation).

SHRMEM - causes the VSP to do instruction fetch from external RAM.

The commands in this last set control the testing functions.

CREATE - makes the VSPS run a test creating input and output files for future validation. It puts the VSPS into a mode which affects the operation of several commands until it is disabled using TEST, or the particular run terminates.

TEST- disables CREATE mode.

DATA DES_FILE DAT_FILE base size - used to generate signals. DES_FILE contains a description of the input data, answering the same prompts as the signal generator in the interactive mode. You need not save this data file; it is created automatically when the VSPS is run in this mode. DAT_FILE is the name of the file where the data is saved. Note that DES_FILE will be read only if DAT_FILE does not exist or if CREATE is set. "base" is the address to write data (also in DES_FILE). "size" is the number of 16-bit words (must be <= size in DES_FILE).

SKIP - disables signal generation by the VSPS if it is in CREATE mode and the DAT_FILE already exists.

NOSKIP - forces signal generation by the VSPS if it is in CREATE mode, even if a copy of the file being created already exists.

COMPAR DAT_FILE base1 size - compares data located at address "base1" in VSP external ram with DAT_FILE. If the VSPS is in CREATE mode, COMPAR creates a file named DAT_FILE, by writing "size" words of the VSP external RAM starting from address "base1".

MCOMPAR base1 base2 size max_err - compares data located at address "base1" in VSP external RAM with data located at address "base2" in VSP external RAM. This is done for "size" comparisons. An error is reported if the maximum difference exceeds max_err.

WRTCMP - puts the VSPS in a mode to write the VSP external RAM to a file when a COMPAR is executed. The format of the file is the same as that for COMPAR in CREATE mode. The name of this file is formed by appending "cr#" to the prefix of the name in COMPAR. "#" is the QOPT number (0, 1, 2 or 3) if you have only hardware queueing enabled; or B, C or D if you have ALLQ enabled and QOPT set to 1, 2 or 3. For example, if DAT_FILE in the COMPAR command was "test.cmp", and you had executed a WRTCMP previous to this command, with ALLQ and QOPT 3, you would get "test.crD" as the output file.

SKIPCMP-disables WRTCMP mode.

COUNTERR - causes the VSPS to count errors but not list them in the VALDTE.ERR file.

LISTERR - causes the VSPS to list errors, and give a total error count in the VALDTE.ERR file.

TIME TIM_FILE - causes the VSPS to check the execution time since the last reset of the clock, or since the VSPS was started against the contents of TIM_FILE. The clock is reset after the comparison is finished. In CREATE mode, the VSPS creates a file which contains these numbers. There are three numbers: the first is the number of clock ticks since reset, the second is the number of instructions executed, and the third is the number of clock ticks that the bus was in use.

QTIME QTIM_FILE - does the same as TIME except that the file name has the queue option symbol appended to its prefix. This symbol is the same as the "#" described for WRTCMP. For example: if QTIM_FIL is given as "test.tim", and the VSPS is running with QOPT 1 and HWQO, the time file generated would be "test1.tim". If the VSPS is running with QOPT 2 and ALLQ it would be "testC.tim".

CRETIM - puts the VSPS in a mode to create only new time files, not new data files.

CKTIM-disables CRETIM mode.

D.7.2 Validation Example

To validate an algorithm with the VSPS, you must first generate the code you want to check by compiling a new version of the VSPS, as described in Chapter IX. This example uses the program first described in Section 9.2. If you follow the steps in Chapter IX, you will end up with your private copy of the VSPS containing that program. Here are the batch verification mode command files.

file1:"batch1.mac"

```
MSGLEV2
VISIBLE
CREATE
NAMEcreateit
BRANCHbatch2.mac
TEST
NAMEtestit
BRANCHbatch2.mac
```

file2:"batch2.mac"

```
FFPOP
COMPARbtest.cmp0256
TIMEbtest.tim
```

Next, run this copy of the VSPS, and select option M-9. When it prompts you for a file, type in <batch1.mac>, and the VSPS will pick up the first file. Assuming you have compiled a new version of the VSPS with the simple example in FFPOP, this will run a batch job which first generates test data, then runs the program again to check the results. Ordinarily you would not use the create and test modes on the same data in the same run; they are shown here for illustration only.

D.8 The ZRCKMSG Subroutine

Like the ZRHSTBU and ZRSETSYS subroutines described in the Chapter IX, the subroutine ZRCKMSG can be called from within your program. This subroutine enables you to reset the message level of the VSPS. Recall that the message level determines the amount of instruction printout during execution. It is useful to call this subroutine at critical points in the program so that you can skip over those parts of the program you are not interested in. The declaration in the VSPS for ZRCKMSG is:

```
zrckmsg(level)
intlevel;
  {
  };
```

The call to the subroutine from your program is

```
zrckmsg(level);
```

where level is an int variable. The level variable sets the message level, which can be any number from 0 through 3 or 10 through 17. This call will activate the corresponding option in Menu M-4.

Note that you can plot external RAM by calling this subroutine with level = 10.

APPENDIX E

REFERENCES

1. Brigham, E. Oran. The Fast Fourier Transform, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.

2. Oppenheim, A.V. and Schafer, R.W. Digital Signal Processing, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.

3. Rabiner, L.R. and Gold, Bernard. Theory and Application of Digital Signal Processing, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.

4. Kernighan, Brian W., and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Inc, Englewood Cliffs, NJ, 1978.

APPENDIX F

GLOSSARY

ALU	Arithmetic and logic unit. That part of a computer processor performing arithmetic, logical, and related operations.
binary	Pertaining to a selection or condition that has two possible values or states. Pertaining to a number system with a base of two.
bit	Binary digit. A 1 or a 0 in binary arithmetic. A unit of data in the binary numbering system.
buffer	An assigned block of memory for temporarily holding data.
bus	A set of conductors for transmitting signals or power.
butterfly	An element of the computation of a Fourier transform in which two data elements are transformed by complex multiplication and addition into two new data elements in a different domain.
byte	A fixed number of consecutive binary digits, (usually eight) operated on as a unit.
clock cycle	one pulse of the output of a device that generates periodic signals for timing and synchronization. In this manual a clock cycle is 100 nanoseconds.
C	A high-level computer programming language characterized by economy of expression, modern control flow and data structures, and a rich set of operators.
compiler	A program that translates a program in one language to a different language, frequently executable machine code.
CPU	Central processing unit of a computer. It contains the arithmetic and logic unit (ALU), main storage and special register groups.
DFT	Discrete Fourier transform.
DIT	Decimation-in-time.
DIP	Dual in-line package.
DSP	Digital signal processing.
environment	The setting, consisting of the computer and the operating system, in which a software program operates.
external	In this manual, external means outside the VSP or VSPS.
FFT	Fast Fourier transform.

high-level language A programming language that is problem-oriented rather than machine-oriented.

host The main computer system in which a VSP system is installed, or in which the VSPS runs.

IDFT Inverse discrete Fourier transform.

IFFT Inverse fast Fourier transform.

interrupt A hardware or software device that can suspend a process when triggered by an outside event, and does this in such a way that the process can be resumed.

LSB Least significant bit.

MSB Most significant bit.

nibble A group of four consecutive binary digits, usually half a byte, forming a usable data element.

parameter As applied to a VSP instruction: a named element of the instruction to which values are assigned that direct and control the execution of the instruction.

preprocessor A program that translates preliminary setup code, pseudo-operation code, or other special instructions into a form for compilation by the principal compiler.

RAM Random access memory. Computer memory which can be read or written to directly by specifying an address.

reset To cause a counter, flag or value to revert to an initial state or value. The reverse of "set".

ROM Read only memory. Computer memory with fixed information that can be read but not changed.

set To cause a counter, flag or value to take a specified state or value.

ULTRIX The Digital Equipment Corporation proprietary version of UNIX.

UNIX A widely used proprietary operating system originally created at AT&T Bell Laboratories.

VAX The proprietary name for a series of minicomputers manufactured by Digital Equipment Corporation.

VDT Video display terminal.

VMS A proprietary operating system of Digital Equipment Corporation.

VSP Vector Signal Processor. The proprietary name of a product of Zoran Corporation.

VT100 The proprietary name of a widely used video display terminal of Digital Equipment Corporation.

VT240 The proprietary name of a video display terminal of Digital Equipment Corporation with advanced graphics capability.

word A character-string or bit-string that it is convenient for some purpose to consider as an entity. On the VAX and in the VSPS software, a word is two 8-bit bytes, or 16 bits.

ADDENDUM

VSP161 SIMULATOR PLUS EVALUATION BOARD DRIVER (VSPE)

INDEX

ADDENDUM

1. Introduction

The VSP Simulator with the addition of the board driver interface can perform virtually all of its operations directly on the VSP Evaluation Board. This includes signal and data generation, external memory plotting and listing, execution of programs in the signal processing library and execution of user programs.

This supplement assumes that you are thoroughly familiar with the VSPS. The Evaluation Board Driver consists of additional files that are linked with the VSPS to support execution of VSP161 programs on either the Simulator or the Evaluation Board. You can switch back and forth between these two modes of execution. As you switch physically, different memory is used to model the external memory of the VSP161 Systems Processor. All simulator functions, library programs and user programs will be changed to point to either of these two separate memory regions. In contrast, internal VSP memory and registers are treated differently in each of the two. Separate commands and menu items are available to access the internal memory and registers of the simulated and actual VSP161 Systems Processor. The Instruction Tutorial always runs on the VSP161 Simulator.

2. Using the VSP161 Simulator

In default mode, the VSPE begins execution of programs on the simulator. All items in the menus are the same as they are in the standard simulator, with the exception of option '8' in the Main Menu. Instead of allowing execution of user programs, as in the VSPS, this option now invokes a new menu for controlling the Evaluation Board. A user program can be invoked by selecting option '16' in this new menu (Menu M-8). Options 11 and 12 in Menu M-8 allow the user to choose execution on either the Evaluation Board or the Simulator.

All of the standard macro commands are available on the Evaluation Board and function as they do in the VSP161 Simulator. In addition, there are a number of new commands that are tied directly to board execution and should NOT be used unless the VSPE is set to run on the Evaluation Board.

Commands that can be used in either simulation mode or evaluation board mode are:

disasm	-disassemble instructions in memory
DOS	-temporarily return to DOS

MBF	-fill memory region with specified constant
Mdisp	-provide a condensed HEX dump
MIF	-interactively fill memory

Note that the macro command-language accepts upper or lower case letters. It also allows abbreviation of the commands the the shortest unambiguous name. See Chapter VII of this manual for general information on the macro command-language and the User's Manual VSP Debugger Software for additional information on macros not included in the VSPS.

3. Using the VSP161 Evaluation Board

To use the VSPE with the Evaluation Board, select option '11' in Menu M-8. This will cause two changes in the internal VSPE state. First, all accesses to VSP memory will be changed to point to the evaluation board memory. (The only exceptions are commands to move data between the two memories. Such commands will work correctly regardless of which mode VSPE is set to.) The second change is that all VSP instruction execution will be directed to the VSP Evaluation Board. The only exception is the Instruction Tutorial, which always runs on the Simulator.

The second line in the Evaluation Board Display Options Menu (M-8) indicates if VSPE is set to run with the board or with the Simulator.

A possible source of confusion is the different way that internal and external memory are treated when you switch to the Evaluation Board. Commands that access external memory will all use board memory. Commands that access internal memory will not be changed. Only menu options in menus M-8 and M-8-19 access internal memory and registers on the VSPE board. All other commands and menus ALWAYS access the simulated internal registers and memory.

3.1 VSP161 Speed and Memory Mapped Reads

The VSPE evaluation board mode contains two submodes which are dependent on the way in which the board is configured. If you have a 15MHz crystal, then memory mapped reads must not be used to interrogate any of the VSP161 registers. **Doing so will cause program failure!**

The one exception to this is the status register. Memory mapped reads may be used to clear the interrupt bits in this register. If you are using a 10MHz crystal, the memory mapped reads are fully supported and can be used. (This limitation of the VSP161 will be removed in the C stepping.) The VSPE automatically measures the speed of the board you are using and sets the memory mapped flag accordingly. To ensure that this test runs correctly, do not type ahead when starting execution of VSPE--wait until it prompts for input. The third line of Menu M-8 indicates if memory mapped reads are used when the VSPE is set to run on the evaluation board.

The implication of not having memory mapped reads is that it is not possible to determine any internal state information unless the VSP is halted. In particular, it is not possible to read the fetch address unless the VSP is halted. Because of this, there are only two reliable methods for insuring that a VSP program has halted when executing in this mode. You may either wait long enough to be certain that the program has halted (time out) or you must disable all interrupts by setting the VSP161 mode register appropriately and by setting the EI bit in the last instruction of the program (the EI bit is not maskable). Because of the VSP161 instruction overlap capability, special precautions must be taken if the mode register is set for two RAM sections. In this case, the last two instructions in the program should be NOPs with the RS fields set to 0 and 1, respectively. The last NOP should have the EI bit set.

3.2 Determining If the VSP161 Is Halted

In the C stepping of the VSP161, the ILI interrupt can only occur when the processor is completely idle. In this stepping, memory mapped reads will be supported at full processor speed. Because of errors in the B stepping, there are a number of different tests for VSP161 idle that are applicable, depending on the execution mode. The following table summarizes these tests when using the VSP161 instruction fetch feature.

	Memory Mapped Reads	No Memory Mapped Reads
Interrupt on Completion Only	Quit on any Interrupt	Quit on any Interrupt
All Interrupts Enabled	Quit on ILI Interrupt and Fetch Address at HLT	Always Time Out

By using options 7 and 8 in Menu M-8-19 the user can inform the VSPE if the programs are set up to interrupt ONLY on completion or whether additional interrupts may occur.

3.3 IMMEDIATE Execution

The DEFER and IMMEDIATE commands of the VSP parser work on the board in the same way as they do on the VSPS. In IMMEDIATE mode, each instruction is written to the VSP161 queue. The AT then goes into a WAIT loop until and ILI interrupt occurs. MAKE SURE THAT THIS INTERRUPT IS NOT MASKED WHEN EXECUTING INSTRUCTIONS IN THIS MODE OR THERE WILL BE A TIME OUT ON EVERY INSTRUCTION. If a JMPI is encountered when running in this mode, the protocol for controlling the VSP161 changes to deal with DEFERred instructions.

3.4 Executing DEFERred Instructions

The protocol for DEFERred instructions is a function of whether memory mapped reads are supported. If they are not supported, execution is assumed to have terminated only if an interrupt occurs and none of the maskable status register interrupt flags are set. Such an interrupt can only be generated by setting the EI bit in an instruction. If this is not done, execution will time out. Thus, the first thing the user should do in a block of deferred instructions is to clear all interrupts in the mode register.

Note: this is in contrast to the requirement that the ILI interrupt be enabled in IMMEDIATE mode.

If memory mapped reads are supported, then the protocol assumes that the VSP Systems Processor has halted only if the ILI bit is set and the instruction at the address of the last instruction fetched (as determined by reading the fetch address register) is a HLT. This test only works if the ILI interrupt has been enabled. Thus, when running programs in this mode the user must enable the ILI interrupt.

3.5 Instruction Queueing Not Supported

The various queueing options of the VSPS are not supported with the Evaluation Board or the B stepping of the VSP161. However, all parser statements and simulator options associated with queueing can be used. Code will execute correctly, but the increased efficiency from queueing will not be realized. Note: the various timing options associated with queueing were intended to model the asynchronous behavior of the host and VSP161. Thus, when running with a real host (the AT) and the VSP161 processor, they are no longer meaningful and have no effect.

3.6 Message Levels

Executing instructions on the evaluation board is similar to executing them on the VSPS. The message levels set in either Menu M-4 and Menu M-8 determine what information is displayed. At message level 0, nothing other than error messages or the decision made with regard to interrupt processing are shown. At message level 1, each instruction is shown disassembled. In addition, the contents of the status register before and after each instruction execution are shown.

Finally, the contents of the fetch registers are displayed. JMPI behaves very differently on the two systems. On the board, the instructions executed after the JMPI will not be displayed. Only the fetch and status registers at interrupts or termination of execution will be shown. This is true regardless of the message level. In order to single step DEFERred instructions, the user should use the macro command 'step'.

The information displayed at message level 3 is much less detailed than that available when running on the VSPS. For those instructions that load, store or modify VSP RAM, the contents of the portion of VSP RAM affected are displayed after instruction completion. Similarly, the accumulators are displayed after instructions that modify them. Other instructions have no additional information displayed at message level 3. The contents of RAM before execution is not displayed (although in general it will be displayed from the "output" on a previous instruction).

Note that on the VSPS when running at message level 2, entering any number causes the associated option in Menu M-4 to be executed. When running on the board, the number entered selects an option from Menu M-8. Message level settings 0-3 work the same in both of these menus. Other options, however, operate differently.

In addition, the user must be aware that the Break Menu does not change as a function of VSPE mode. The message levels set by either Menu M-8 or the Break Menu apply equally to execution on the board and execution on the Simulator.

3.7 Macro Command-Language

There are several macro commands that must **only be used when executing on the Evaluation Board.** Erroneous and unpredictable results can be expected if these command are executed in VSPE when set in the Simulator mode. These commands are:

breakpoint	-set a break point in VSP code
continue	-continue execution after break point

compare	-compare memory regions (between or within memory types)
Mmove	-move data blocks (between or within memory types)
run	-execute instructions in VSP memory
step	-single step instructions in VSP memory
test	-check Evaluation Board

There are additional macros that affect only the Evaluation Board but will execute correctly regardless of VSPE mode. These are:

reset	-reset the Evaluation Board
Rdisp	-display VSP161 registers

4. Switching Between the Evaluation Board and the VSP Simulator

In some cases it may be useful to switch between execution on the Simulator and exectution on the board. In such cases it is advisable that the user create his program and data to both board memory and Simulator external memory. This can be done through options available in the debugging menu, M-8-19.

5. Menu Descriptions

Below is a brief description of the options availabe on menus that are unique to the VSPE and are not available on the VSPS.

5.1 Evaluation Board Driver Display Options (Menu M-8)

Options 0 through 4 in this menu set the message level. Options 5 through 9 are reserved for future expansion. Option '10' displays a HELP file. Option'11' sets VSPE to run programs on the Evaluation Board. Option '12' sets VSPE to run programs on the Simulator. Option '13' displays VSP internal RAM. This display is scaled for the 17-bit integers in the VSP memory. However, the values displayed are those that follow after the automatic unbiased rounding of the least significant bit. This rounding is done whenever the memory is read, and there is no non-destructive way to know the actual value of all 17 bits. There is an option in Menu M-8-19 to read destructively all 17 significant bits plus the two overflow bits of VSP161 RAM.

Option '14' in Menu M-8 displays the VSP161 registers, including the accumulators and the registers used by the SCL instruction. Option '15' will allow execution to begin at any location in memory. A valid program must exist in memory at these locations or the VSP161 will hang up and need to be reset.

Options 17 and 18 go to standard simulator for displaying memory and controlling display options. Internal memory displays arrived at through these options always refer to Simulator memory. Option '19' branches to additional debugging commands.

5.2 Debugging Commands (Menu M-8-19)

The first two options in this menu copy Evaluation Board memory to and from Simulator external memory. This is useful for running the same program with the same data on both the Simulator and the Evaluation Board. Option '4' resets the Evaluation Board and the VSP161 Systems Processor. Option '5' is only available when set to run on the Evaluation Board. It displays all 19 bits of VSP161 internal RAM. To do this, it runs a short program that changes both the mode register and RAM contents.

Option '6' allows the user to adjust the size of the time out counter limit. Setting this too high will cause unnecessary delays. Setting it too low may take the bus from VSP control before a program has completed. Options 7 and 8 allow the user to inform the VSPE if the user programs are set to generate an interrupt ONLY at program completion. If they are not set up this way, and memory mapped reads are not supported, then the only reliable way to insure program completion is to do a time out.